VISIT…

Structured Computer Organization

Fourth Edition

Andrew S. Tanenbaum
With contributions from James R. Goodman

Structured Computer Organization is the fourth edition of this best-selling introduction to computer hardware and architecture. It has been heavily revised to reflect the latest changes in the rapidly changing computer industry. Professor Tanenbaum has maintained his popular method of presenting the computer as a hierarchy of levels, each with a well-defined function. The book is written in a style and level of detail that covers all the major areas, but is still accessible to a broad range of readers.

After an introductory chapter and a chapter about system organization (processors, memories, and I/O devices), we come to the core of the book: chapters on the digital logic level, the microarchitecture level, the instruction set architecture level, the operating system level, and the assembly language level. Finally, there is a chapter on the increasingly important topic of parallel computer architectures.

New to this edition:

- The running examples throughout the book are now the Pentium II, Sun UltraSPARC II, and Java Virtual Machine, inlcuding Sun's hardware Java chip that implements JVM

- The input/output devices discussed in the **computer systems organization** chapter have been updated to emphasize modern technology such as RAID disks, CD-recordables, DVD, and color printers

- Modern computer buses, such as the PCI bus and USB bus have been added to the **digital logic level**.

- The microarchitecture level has been completely rewritten and updated as follows:

    - The detailed microprogrammed machine example illustrating data path control is now based on a subset of the Java Virtual Machine
    - Design cost and performance tradeoffs are illustrated with a series of detailed examples culminating with the Mic-4 example that uses a seven-stage pipeline to introduce how modern computers such as the Pentium II work
    - A new section on improving performance focuses on the most recent techniques such as caching, branch prediction, out-of-order execution, and speculative execution

- The **instruction architecture level** covers machine language using the new running examples.
- The **operating system level** includes examples for the Pentium II (Windows NT) and the UltraSPARC II (UNIX).
- New material on dynamic linking has been added to the **assembly language level** .
- The **parallel computer architecture** chapter has been completely rewritten and expanded to include a detailed treatment of multiprocessors (UMA, NUMA, and COMA), and multicomputers (MPP and COW).
- All code examples have been rewritten in Java.

With nearly 300 end-of-chapter exercises. an up-to-date annotated bibliography, figure files available for downloading, a Mic-1 simulator for student use, and a new instructor's manual, this book is ideal for courses in computer architecture or assembly language programming.

Preface

The first three editions of this book were based on the idea that a computer can be regarded as a hierarchy of levels, each one performing some well-defined function. This fundamental concept is as valid today as it was when the first edition came out, so it has been retained as the basis for the fourth edition. As in the first three editions, the digital logic level, the microarchitecture level, the instruction set architecture level, the operating system machine level, and the assembly language level are all discussed in detail (although we have changed some of the names to reflect modern practice).

Although the basic structure has been maintained, this fourth edition contains many changes, both small and large, that bring it up to date in the rapidly changing computer industry. For example, all the code examples, which were in Pascal, have been rewritten in Java, reflecting the popularity of Java in the computer world. Also, the example machines used have been brought up to date. The current examples are the Intel Pentium II, the Sun UltraSPARC II, and the Sun picoJava II, an embedded low-cost hardware Java chip.

Multiprocessors and parallel computers have also come in widespread use since the third edition, so the material on parallel architectures has been completely redone and greatly expanded, now covering a wide range of topics, from multiprocessors to COWs.

The book has become longer over the years (although still not as long as some other popular books on the subject). Such an expansion is inevitable as a subject develops and there is more known about it. As a result, when the book is used for a course, it may not always be possible to finish the book in a single course (e.g., in a trimester system). A possible approach would be to do all of Chaps. 1, 2, and 3, the first part of Chap. 4 (up through and including Sec. 4.4), and Chap. 5 as a bare minimum. The remaining time could be filled with the rest of Chap. 4, and parts of Chaps. 6, 7, and 8, depending on the interest of the instructor.

A chapter-by-chapter rundown of the major changes since the third edition follows. Chapter 1 still contains an historical overview of computer architecture, pointing out how we got where we are now and what the milestones were along the way. The enlarged spectrum of computers that exist is now discussed, and our three major examples (Pentium II, UltraSPARC II, and picoJava II) are introduced.

In Chapter 2, the material on input/output devices has been updated, emphasizing the technology of modern devices, including RAID disks, CD-Recordables, DVD, and color printers, among many others.

Chapter 3 (digital logic level) has undergone some revision and now treats computer buses and modern I/O chips. The major change here is additional

material on buses, especially the PCI bus and the USB bus. The three new examples are described here at the chip level.

Chapter 4 (now called the microarchitecture level) has been completely rewritten. The idea of using a detailed example of a microprogrammed machine to illustrate the ideas of data path control has been retained, but the example machine is now a subset of the Java Virtual Machine. The underlying microarchitecture has been correspondingly changed. Several iterations of the design are given, showing what trade-offs are possible in terms of cost and performance. The last example, the Mic-4, uses a seven-stage pipeline and provides an easy introduction to how important modern computers, such as the Pentium II, work. A new section on improving performance has been added, focusing on the most recent techniques such as caching, branch prediction, (superscalar) out-of-order execution, speculative execution, and predication. The new example machines are discussed at the microarchitecture level.

Chapter 5 (now called the instruction set architecture level) deals with what many people refer to as *(OQmachine language.*(CQ The Pentium II, UltraSPARC II and Java Virtual Machine are used as the primary examples here.

Chapter 6 (operating system machine level) has examples for the Pentium II (Windows NT) and UltraSPARC II (UNIX). The former is new and has many features that are worth looking at, but UNIX is still a reliable workhorse at many universities and companies and is well worth examining in detail as well due to its simple and elegant design.

Chapter 7 (assembly language level) has been brought up to date by using examples from the machines we have been studying. New material on dynamic linking has been added as well.

Chapter 8 (parallel computer architectures) has been completely rewritten from the third edition. It now covers both multiprocessors (UMA, NUMA, and COMA) in detail, as well as multicomputers (MPP and COW).

The bibliography has been extensively revised and brought up to date. Well over two-thirds the references refer to works published after the third edition was published. Binary numbers and floating-point numbers have not undergone much change recently, so the appendices are largely the same as in the previous edition.

Finally, some problems have been revised and many new problems have been added since the third edition. Accordingly, a new problem solutions manual is available from Prentice Hall. It is available .I only to faculty members, who can request a free copy from their Prentice Hall representative.

A Web site for this book is available. PostScript files for all the illustrations used in the book are available electronically. They can be fetched and printed, for example, for making overhead sheets. In addition, a simulator and other and software tools are there too. The URL for this site is .HS .ti 0.25i \fIhttp://www.cs.vu.nl/\(tiast/sco4/\fP .HS The simulator and software tools

were produced by Ray Ontko. The author wishes to express his gratitude to Ray for producing these extremely useful programs.

A number of people have read (parts of) the manuscript and provided useful suggestions or have been helpful in other ways. In particular, I would like to thank Henri Bal, Alan Charlesworth, Stan Eisenstat, Kourosh Gharachorloo, Marcus Goncalves, Karen Panetta Lentz, Timothy Mattson, Harlan McGhan, Miles Murdocca, Kevin Normoyle, Mike O'Connor, Mitsunori Ogihara, Ray Ontko, Aske Plaat, William Potvin II, Nagarajan Prabhakaran. James H. Pugsley, Ronald N. Schroeder, Ryan Shoemaker, Charles Silio, Jr., and Dale Skrien for their help, for which I am most grateful. My students, especially Adriaan Bon, Laura de Vries, Dolf Loth, Guido van 't Noordende, have also helped debug the text. Thank you.

I would especially like to thank Jim Goodman for his many contributions to this book, especially to Chaps. 4 and 5. The idea of using the Java Virtual Machine was his, as were the microarchitectures for implementing it. Many of the advanced ideas were due to him. The book is far better for his having put in so much effort.

Finally, I would like to thank Suzanne for her patience for my long hours in front of my Pentium. From my point of view the Pentium is a big improvement over my older 386 but from hers, it does not make much difference. I also want to thank Barbara and Marvin for being great kids and Bram for always being quiet when I was trying to write.

Andrew S. Tanenbaum

© 2000-2001 by Prentice-Hall, Inc.
A Pearson Company
Distance Learning at Prentice Hall
Legal Notice

Book Resources

Table of Contents

Chapter 1 **INTRODUCTION 1**

1.1 **Structured Computer Organization** 2
- 1.1.1 Languages, Levels, and Virtual Machines 2
- 1.1.2 Contemporary Multilevel Machines 4
- 1.1.3 Evolution of Multilevel Machines 8

1.2 **Milestones In Computer Architecture** 13
- 1.2.1 The Zeroth Generation-Mechanical Computers (1642-1945) 13
- 1.2.2 The First Generation-Vacuum Tubes (1945-1955) 16
- 1.2.3 The Second Generation-Transistors (1955-1965) 19
- 1.2.4 The Third Generation-Integrated Circuits (1965-1980) 21
- 1.2.5 The Fourth Generation-Very Large Scale Integration (1980-?) 23

1.3 **The Computer Zoo** 24
- 1.3.1 Technological and Economic Forces 25
- 1.3.2 The Computer Spectrum 26

1.4 **Example Computer Families** 29
- 1.4.1 Introduction to the Pentium II 29
- 1.4.2 Introduction to the UltraSPARC II 31
- 1.4.3 Introduction to the picoJava II 34

1.5 **Outline Of This Book** 36

Chapter 2 **COMPUTER SYSTEMS ORGANIZATION 39**

2.1 **Processors** 39
- 2.1.1 CPU Organization 40
- 2.1.2 Instruction Execution 42
- 2.1.3 RISC versus CISC 46
- 2.1.4 Design Principles for Modern Computers 47
- 2.1.5 Instruction-Level Parallelism 49
- 2.1.6 Processor-Level Parallelism 53

2.2 **Primary Memory** 56

2.2.1 Bits 56
2.2.2 Memory Addresses 57
2.2.3 Byte Ordering 58
2.2.4 Error-Correcting Codes 61
2.2.5 Cache Memory 65
2.2.6 Memory Packaging and Types 67

2.3 **Secondary Memory** 68
2.3.1 Memory Hierarchies 69
2.3.2 Magnetic Disks 70
2.3.3 Floppy Disks 73
2.3.4 IDE Disks 73
2.3.5 SCSI Disks 75
2.3.6 RAID 76
2.3.7 CD-ROMs 80
2.3.8 CD-Recordables 84
2.3.9 CD-Rewritables 86
2.3.10 DVD 86

2.4 **Input/Output** 89
2.4.1 Buses 89
2.4.2 Terminals 91
2.4.3 Mice 99
2.4.4 Printers 101
2.4.5 Modems 106
2.4.6 Character Codes 109

2.5 **Summary** 113

Chapter 3 **THE DIGITAL LOGIC LEVEL 117**

3.1 **Gates And Boolean Algebra** 117
3.1.1 Gates 118
3.1.2 Boolean Algebra 120
3.1.3 Implementation of Boolean Functions 122
3.1.4 Circuit Equivalence 123

3.2 **Basic Digital Logic Circuits** 128
3.2.1 Integrated Circuits 128
3.2.2 Combinational Circuits 129
3.2.3 Arithmetic Circuits 134
3.2.4 Clocks 139

3.3 **Memory** 141

3.3.1 Latches 141
3.3.2 Flip-Flops 143
3.3.3 Registers 145
3.3.4 Memory Organization 146
3.3.5 Memory Chips 150
3.3.6 RAMs and ROMs 152

3.4 **Cpu Chips And Buses** 154
3.4.1 CPU Chips 154
3.4.2 Computer Buses 156
3.4.3 Bus Width 159
3.4.4 Bus Clocking 160
3.4.5 Bus Arbitration 165
3.4.6 Bus Operations 167

3.5 **Example Cpu Chips** 170
3.5.1 The Pentium II 170
3.5.2 The UltraSPARC II 176
3.5.3 The picoJava II 179

3.6 **Example Buses** 181
3.6.1 The ISA Bus 181
3.6.2 The PCI Bus 183
3.6.3 The Universal Serial Bus 189

3.7 **Interfacing** 193
3.7.1 I/O Chips 193
3.7.2 Address Decoding 195

3.8 **Summary** 198

Chapter 4 **THE MICROARCHITECTURE LEVEL 203**

4.1 **An Example Microarchitecture** 203
4.1.1 The Data Path 204
4.1.2 Microinstructions 211
4.1.3 Microinstruction Control: The Mic-1 213

4.2 **An Example Isa: Ijvm** 218
4.2.1 Stacks 218
4.2.2 The IJVM Memory Model 220
4.2.3 The IJVM Instruction Set 222
4.2.4 Compiling Java to IJVM 226

4.3 **An Example Implementation** 227
4.3.1 Microinstructions and Notation 227
4.3.2 Implementation of IJVM Using the Mic-1 232

4.4 **Design Of The Microarchitecture Level** 243
4.4.1 Speed versus Cost 243
4.4.2 Reducing the Execution Path Length 245
4.4.3 A Design with Prefetching: The Mic-2 253
4.4.4 A Pipelined Design: The Mic-3 253
4.4.5 A Seven-Stage Pipeline: The Mic-4 260

4.5 **Improving Performance** 264
4.5.1 Cache Memory 265
4.5.2 Branch Prediction 270
4.5.3 Out-of-Order Execution and Register Renaming 276
4.5.4 Speculative Execution 281

4.6 **Examples Of The Microarchitecture Level** 283
4.6.1 The Microarchitecture of the Pentium II CPU 283
4.6.2 The Microarchitecture of the UltraSPARC-II CPU 288
4.6.3 The Microarchitecture of the picoJava II CPU 291
4.6.4 A Comparison of the Pentium, UltraSPARC, and picoJava 296

4.7 **Summary** 298

Chapter 5 **THE INSTRUCTION SET ARCHITECTURE LEVEL 303**

5.1 **Overview Of The Isa Level** 305
5.1.1 Properties of the ISA Level 305
5.1.2 Memory Models 307
5.1.3 Registers 309
5.1.4 Instructions 311
5.1.5 Overview of the The Pentium II ISA Level 311
5.1.6 Overview of the The UltraSPARC II ISA Level 313
5.1.7 Overview of the Java Virtual Machine 317

5.2 **Data Types** 318
5.2.1 Numeric Data Types 319
5.2.2 Nonnumeric Data Types 319
5.2.3 Data Types on the Pentium II 320
5.2.4 Data Types on the UltraSPARC II 321
5.2.5 Data Types on the Java Virtual Machine 321

5.3 **Instruction Formats** 322
5.3.1 Design Criteria for Instruction Formats 322
5.3.2 Expanding Opcodes 325
5.3.3 The Pentium II Instruction Formats 327
5.3.4 The UltraSPARC II Instruction Formats 328
5.3.5 The JVM Instruction Formats 330

5.4 **Addressing** 332
5.4.1 Addressing Modes 333
5.4.2 Immediate Addressing 334
5.4.3 Direct Addressing 334
5.4.4 Register Addressing 334
5.4.5 Register Indirect Addressing 335
5.4.6 Indexed Addressing 336
5.4.7 Based-Indexed Addressing 338
5.4.8 Stack Addressing 338
5.4.9 Addressing Modes for Branch Instructions 341
5.4.10 Orthogonality of Opcodes and Addressing Modes 342
5.4.11 The Pentium II Addressing Modes 344
5.4.12 The UltraSPARC II Addressing Modes 346
5.4.13 The JVM Addressing Modes 346
5.4.14 Discussion of Addressing Modes 347

5.5 **Instruction Types** 348
5.5.1 Data Movement Instructions 348
5.5.2 Dyadic Operations 349
5.5.3 Monadic Operations 350
5.5.4 Comparisons and Conditional Branches 352
5.5.5 Procedure Call Instructions 353
5.5.6 Loop Control 354
5.5.7 Input/Output 356
5.5.8 The Pentium II Instructions 359
5.5.9 The UltraSPARC II Instructions 362
5.5.10 The picoJava II Instructions 364
5.5.11 Comparison of Instruction Sets 369

5.6 **Flow Of Control** 370
5.6.1 Sequential Flow of Control and Branches 371
5.6.2 Procedures 372
5.6.3 Coroutines 376
5.6.4 Traps 379
5.6.5 Interrupts 379

5.7 **A Detailed Example: The Towers Of Hanoi** 383
5.7.1 The Towers of Hanoi in Pentium II Assembly Language 384
5.7.2 The Towers of Hanoi in UltraSPARC II Assembly Language 384
5.7.3 The Towers of Hanoi in JVM Assembly Language 386

5.8 **The Intel IA-64** 388
5.8.1 The Problem with the Pentium II 390
5.8.2 The IA-64 Model: Explicitly Parallel Instruction Computing 391
5.8.3 Predication 393
5.8.4 Speculative Loads 395
5.8.5 Reality Check 396

5.9 **Summary** 397

Chapter 6 **THE OPERATING SYSTEM MACHINE LEVEL 403**

6.1 **Virtual Memory** 404
6.1.1 Paging 405
6.1.2 Implementation of Paging 407
6.1.3 Demand Paging and the Working Set Model 409
6.1.4 Page Replacement Policy 412
6.1.5 Page Size and Fragmentation 414
6.1.6 Segmentation 415
6.1.7 Implementation of Segmentation 418
6.1.8 Virtual Memory on the Pentium II 421
6.1.9 Virtual Memory on the UltraSPARC 426
6.1.10 Virtual Memory and Caching 428

6.2 **Virtual I/O Instructions** 429
6.2.1 Files 430
6.2.2 Implementation of Virtual I/O Instructions 431
6.2.3 Directory Management Instructions 435

6.3 **Virtual Instructions For Parallel Processing** 436
6.3.1 Process Creation 437
6.3.2 Race Conditions 438
6.3.3 Process Synchronization Using Semaphores 442

6.4 **Example Operating Systems** 446
6.4.1 Introduction 446
6.4.2 Examples of Virtual Memory 455
6.4.3 Examples of Virtual I/O 459
6.4.4 Examples of Process Management 470

6.5 **Summary** 476

Chapter 7 **THE ASSEMBLY LANGUAGE LEVEL 483**

7.1 **Introduction To Assembly Language** 484
7.1.1 What Is an Assembly Language? 484
7.1.2 Why Use Assembly Language? 485
7.1.3 Format of an Assembly Language Statement 488
7.1.4 Pseudoinstructions 491

7.2 **Macros** 494
7.2.1 Macro Definition, Call, and Expansion 494
7.2.2 Macros with Parameters 496
7.2.3 Advanced Features 497
7.2.4 Implementation of a Macro Facility in an Assembler 498

7.3 **The Assembly Process** 498
7.3.1 Two-Pass Assemblers 498
7.3.2 Pass One 499
7.3.3 Pass Two 502
7.3.4 The Symbol Table 505

7.4 **Linking And Loading** 506
7.4.1 Tasks Performed by the Linker 508
7.4.2 Structure of an Object Module 511
7.4.3 Binding Time and Dynamic Relocation 512
7.4.4 Dynamic Linking 515

7.5 **Summary** 519

Chapter 8 **PARALLEL COMPUTER ARCHITECTURES 523**

8.1 **Design Issues For Parallel Computers** 524
8.1.1 Communication Models 526
8.1.2 Interconnection Networks 530
8.1.3 Performance 539
8.1.4 Software 545
8.1.5 Taxonomy of Parallel Computers 551

8.2 **SIMD Computers** 554
8.2.1 Array Processors 554
8.2.2 Vector Processors 555

8.3 **Shared-Memory Multiprocessors** 559
8.3.1 Memory Semantics 559
8.3.2 UMA Bus-Based SMP Architectures 564
8.3.3 UMA Multiprocessors Using Crossbar Switches 569

8.3.4 UMA Multiprocessors Using Multistage Switching Networks 571
8.3.5 NUMA Multiprocessors 573
8.3.6 Cache Coherent NUMA Multiprocessors 575
8.3.7 COMA Multiprocessors 585

8.4 **Message-Passing Multicomputers** 586
8.4.1 MPPs-Massively Parallel Processors 587
8.4.2 COWs-Clusters of Workstations 592
8.4.3 Scheduling 593
8.4.4 Communication Software for Multicomputers 598
8.4.5 Application-Level Shared Memory 601

8.5 **Summary** 609

Chapter 9 **READING LIST AND BIBLIOGRAPHY 613**

9.1 **Suggestions For Further Reading** 613
9.1.1 Introduction and General Works 613
9.1.2 Computer Systems Organization 614
9.1.3 The Digital Logic Level 615
9.1.4 The Microarchitecture Level 616
9.1.5 The Instruction Set Architecture Level 617
9.1.6 The Operating System Machine Level 617
9.1.7 The Assembly Language Level 618
9.1.8 Parallel Computer Architectures 618
9.1.9 Binary and Floating-Point Numbers 620

9.2 **Alphabetical Bibliography** 620

Appendix A **BINARY NUMBERS** 631

A.1 **Finite-Precision Numbers** 631
A.2 **Radix Number Systems** 633
A.3 **Conversion From One Radix To Another** 635
A.4 **Negative Binary Numbers** 637
A.5 **Binary Arithmetic** 640

Appendix B **FLOATING-POINT NUMBERS** 643

B.1 **Principles Of Floating Point** 644
B.2 **Ieee Floating-Point Standard** 754 646

INDEX 653

1

INTRODUCTION

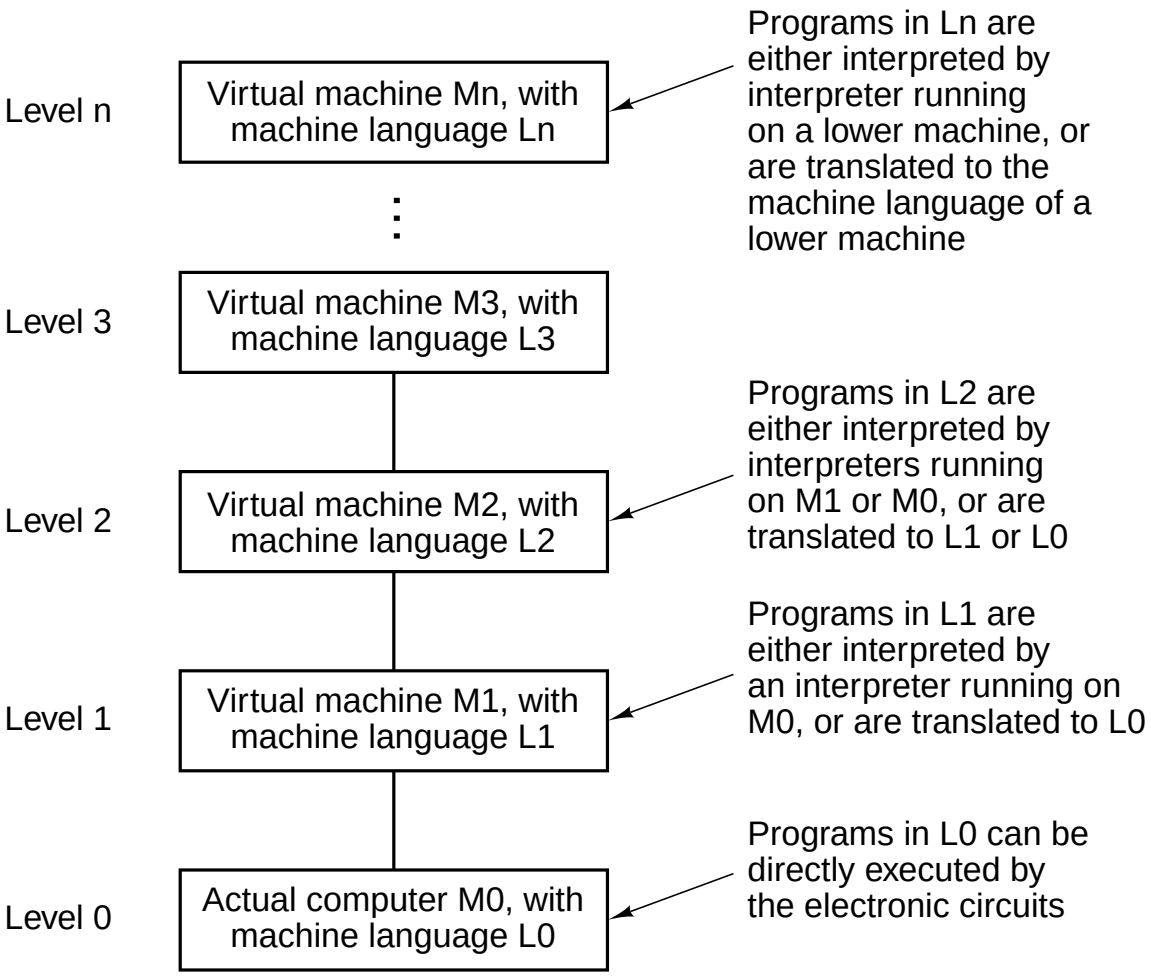

Level n
Virtual machine Mn, with machine language Ln
Programs in Ln are either interpreted by interpreter running on a lower machine, or are translated to the machine language of a lower machine
Level 3
Virtual machine M3, with machine language L3
Level 2
Virtual machine M2, with machine language L2
Programs in L2 are either interpreted by interpreters running on M1 or M0, or are translated to L1 or L0
Level 1
Virtual machine M1, with machine language L1
Programs in L1 are either interpreted by an interpreter running on M0, or are translated to L0
Level 0
Actual computer M0, with machine language L0
Programs in L0 can be directly executed by the electronic circuits

Figure 1-1. A multilevel machine.

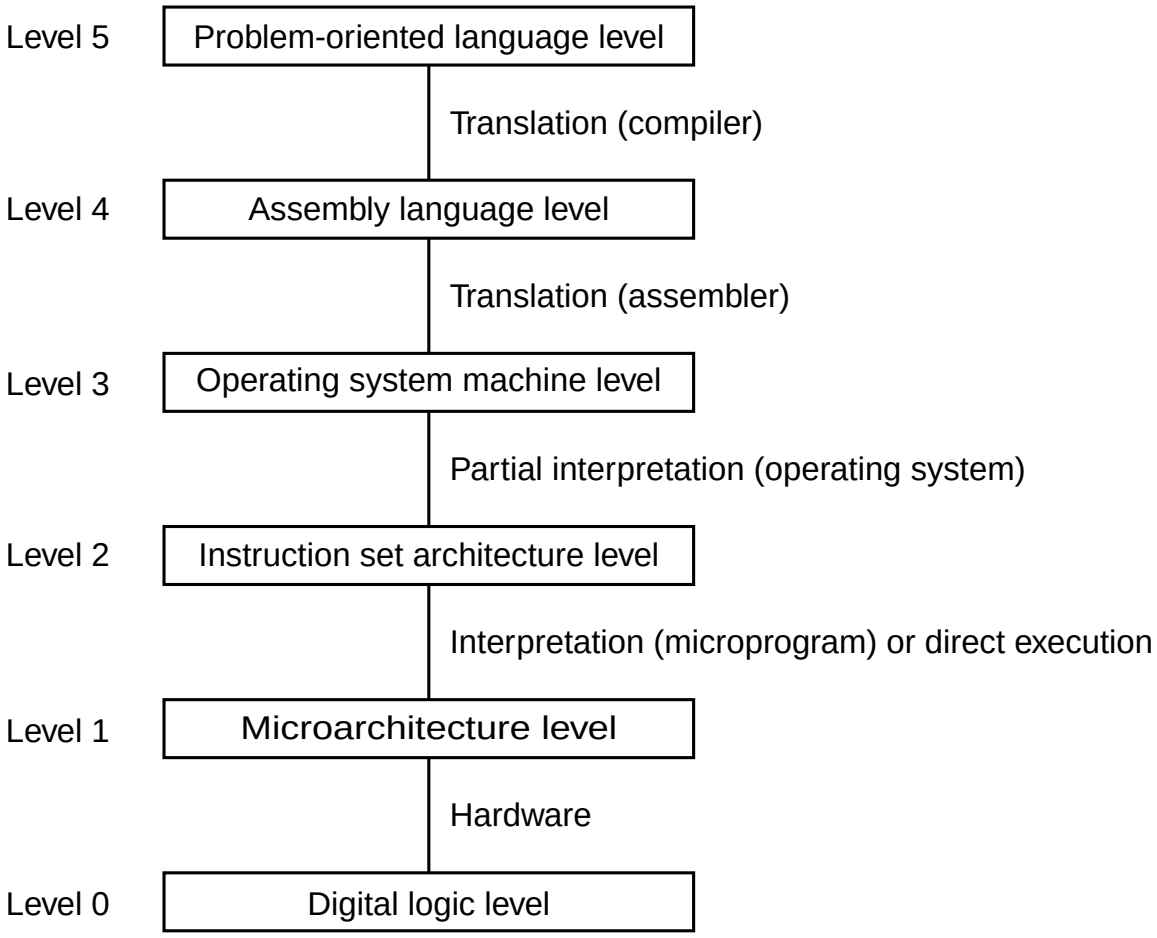


Figure 1-2. A six-level computer. The support method for each level is supported is indicated below it (along with the name of the supporting program).

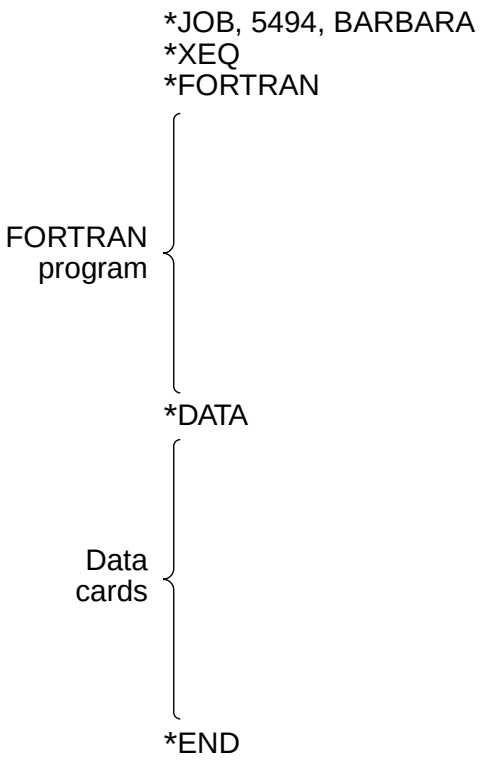


Figure 1-3. A sample job for the FMS operating system.

Year	Name	Made by	Comments
1834	Analytical Engine	Babbage	First attempt to build a digital computer
1936	Z1	Zuse	First working relay calculating machine
1943	COLOSSUS	British gov't	First electronic computer
1944	Mark I	Aiken	First American general-purpose computer
1946	ENIAC I	Eckert/Mauchley	Modern computer history starts here
1949	EDSAC	Wilkes	First stored-program computer
1951	Whirlwind I	M.I.T.	First real-time computer
1952	IAS	Von Neumann	Most current machines use this design
1960	PDP-1	DEC	First minicomputer (50 sold)
1961	1401	IBM	Enormously popular small business machine
1962	7094	IBM	Dominated scientific computing in the early 1960s
1963	B5000	Burroughs	First machine designed for a high-level language
1964	360	IBM	First product line designed as a family
1964	6600	CDC	First scientific supercomputer
1965	PDP-8	DEC	First mass-market minicomputer (50,000 sold)
1970	PDP-11	DEC	Dominated minicomputers in the 1970s
1974	8080	Intel	First general-purpose 8-bit computer on a chip
1974	CRAY-1	Cray	First vector supercomputer
1978	VAX	DEC	First 32-bit superminicomputer
1981	IBM PC	IBM	Started the modern personal computer era
1985	MIPS	MIPS	First commercial RISC machine
1987	SPARC	Sun	First SPARC-based RISC workstation
1990	RS6000	IBM	First superscalar machine

Figure 1-4. Some milestones in the development of the modern digital computer.

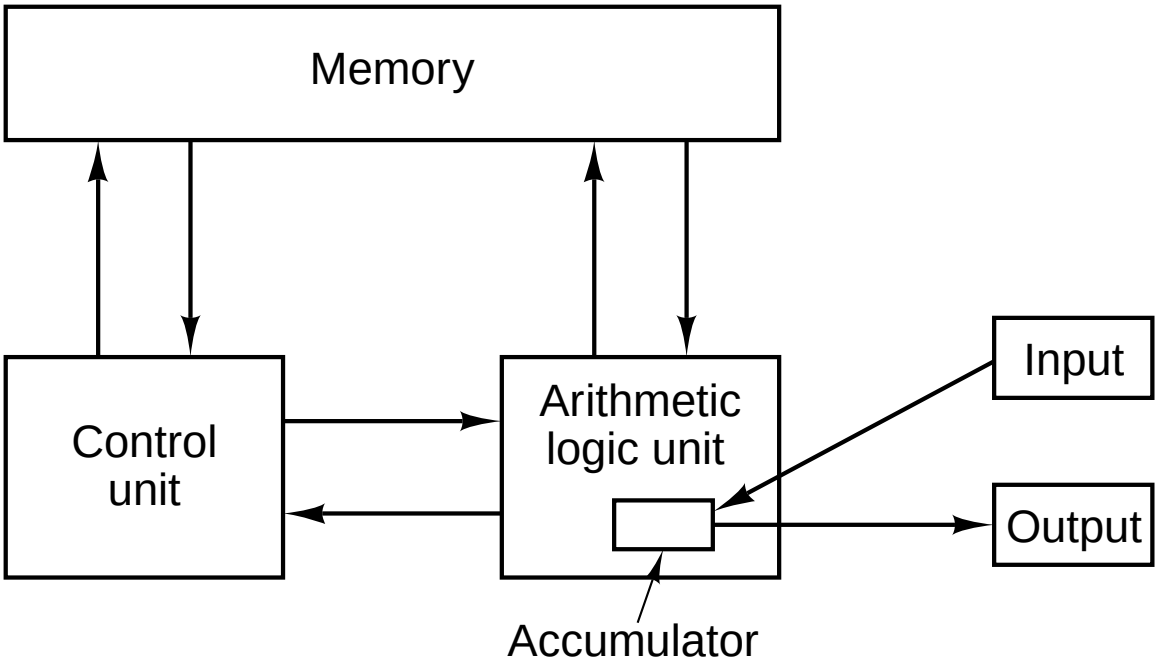


Figure 1-5. The original von Neumann machine.

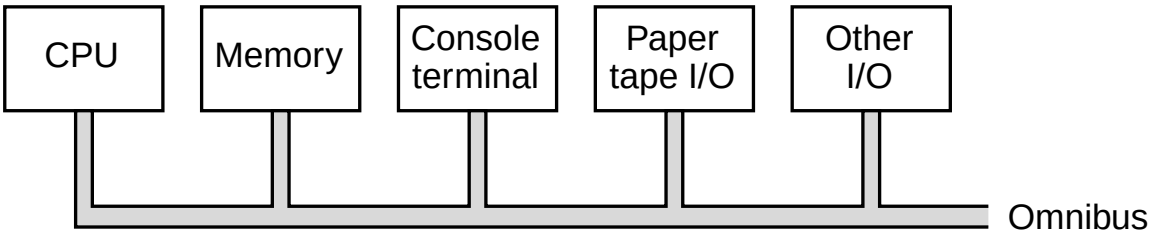


Figure 1-6. The PDP-8 omnibus.

Property	Model 30	Model 40	Model 50	Model 65
Relative performance	1	3.5	10	21
Cycle time (nsec)	1000	625	500	250
Maximum memory (KB)	64	256	256	512
Bytes fetched per cycle	1	2	4	16
Maximum number of data channels	3	3	4	6

Figure 1-7. The initial offering of the IBM 360 product line.

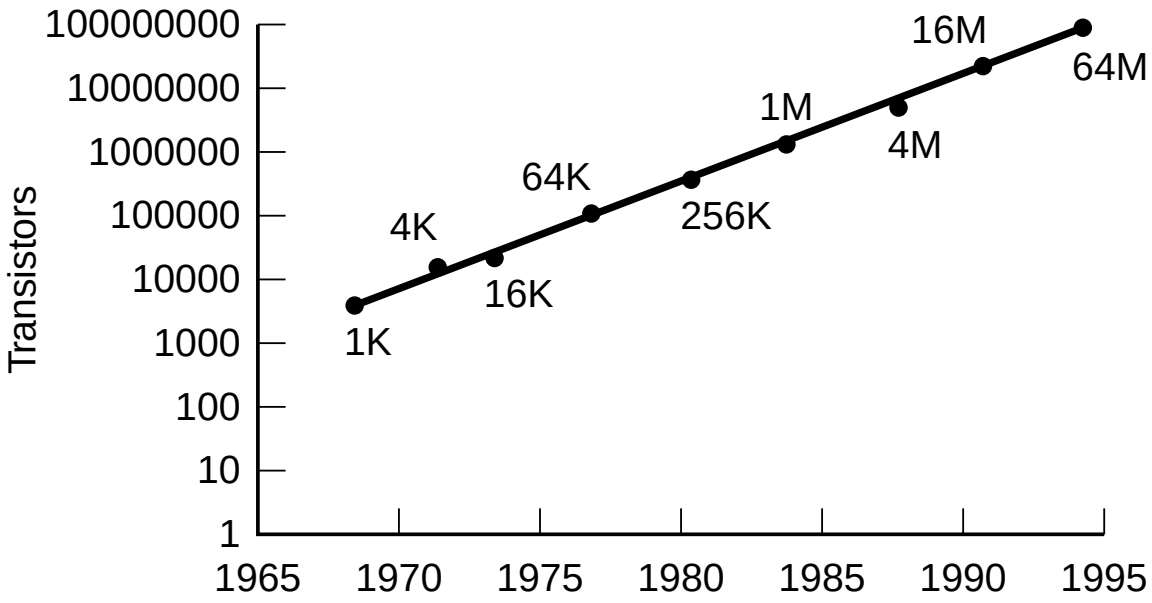


Figure 1-8. Moore's law predicts a 60 percent annual increase in the number of transistors that can be put on a chip. The data points given in this figure are memory sizes, in bits.

Type	Price ($)	Example application
Disposable computer	1	Greeting cards
Embedded computer	10	Watches, cars, appliances
Game computer	100	Home video games
Personal computer	1K	Desktop or portable computer
Server	10K	Network server
Collection of Workstations	100K	Departmental minisupercomputer
Mainframe	1M	Batch data processing in a bank
Supercomputer	10M	Long range weather prediction

Figure 1-9. The current spectrum of computers available. The prices should be taken with a grain (or better yet, a metric ton) of salt.

Chip	Date	MHz	Transistors	Memory	Notes
4004	4/1971	0.108	2,300	640	First microprocessor on a chip
8008	4/1972	0.108	3,500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6,000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5-10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5-8	29,000	1 MB	Used in IBM PC
80286	2/1982	8-12	134,000	16 MB	Memory protection present
80386	10/1985	16-33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25-100	1.2M	4 GB	Built-in 8K cache memory
Pentium	3/1993	60-233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150-200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233-400	7.5M	4 GB	Pentium Pro plus MMX

Figure 1-10. The Intel CPU family. Clock speeds are measured in MHz (megahertz) where 1 MHz is 1 million cycles/sec.

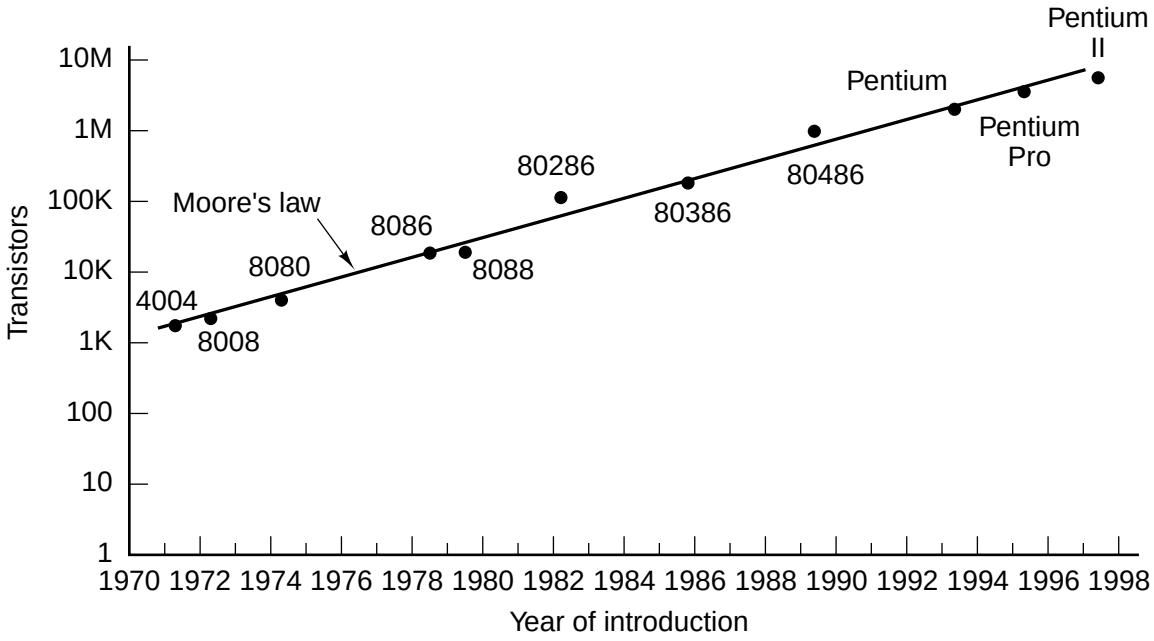


Figure 1-11. Moore’s law for CPU chips.

1.4 EXAMPLE COMPUTER FAMILIES

In this section we will give a brief introduction to the three computers that will be used as examples in the rest of the book: the Pentium II, the UltraSPARC II, and the picoJava II [sic].

1.4.1 Introduction to the Pentium II

In 1968, Robert Noyce, inventor of the silicon integrated circuit, Gordon Moore, of Moore's law fame, and Arthur Rock, a San Francisco venture capitalist, formed the Intel Corporation to make memory chips. In its first year of operation, Intel sold only $3000 worth of chips, but business has picked up since then.

In the late 1960s, calculators were large electromechanical machines the size of a modern laser printer and weighing 20 kg. In Sept. 1969, a Japanese company, Busicom, approached Intel with a request for it to manufacture 12 custom chips for a proposed electronic calculator. The Intel engineer assigned to this project, Ted Hoff, looked at the plan and realized that he could put a 4-bit general-purpose CPU on a single chip that would do the same thing and be simpler and cheaper as well. Thus in 1970, the first single-chip CPU, the 2300-transistor 4004 was born (Faggin et al., 1996).

It is worth noting that neither Intel nor Busicom had any idea what they had just done. When Intel decided that it might be worth a try to use the 4004 in other projects, it offered to buy back all the rights to the new chip from Busicom by returning the $60,000 Busicom had paid Intel to develop it. Intel's offer was quickly accepted, at which point it began working on an 8-bit version of the chip, the 8008, introduced in 1972.

Intel did not expect much demand for the 8008, so it set up a low-volume production line. Much to everyone's amazement, there was an enormous amount of interest, so Intel set about designing a new CPU chip that got around the 8008's 16K memory limit (imposed by the number of pins on the chip). This design resulted in the 8080, a small, general-purpose CPU, introduced in 1974. Much like the PDP-8, this product took the industry by storm and instantly became a mass market item. Only instead of selling thousands, as DEC had, Intel sold millions.

In 1978 came the 8086, a true 16-bit CPU on a single chip. The 8086 was designed to be somewhat similar to the 8080, but it was not completely compatible with the 8080. The 8086 was followed by the 8088, which had the same architecture as the 8086, and ran the same programs but had an 8-bit bus instead of a 16-bit bus, making it both slower and cheaper than the 8086. When IBM chose the 8088 as the CPU for the original IBM PC, this chip quickly became the personal computer industry standard.

Neither the 8088 nor the 8086 could address more than 1 megabyte of memory. By the early 1980s this became more and more of a serious problem, so

Intel designed the 80286, an upward compatible version of the 8086. The basic instruction set was essentially the same as that of the 8086 and 8088, but the memory organization was quite different, and rather awkward, due to the requirement of compatibility with the older chips. The 80286 was used in the IBM PC/AT and in the midrange PS/2 models. Like the 8088, it was a huge success, mostly because people viewed it as a faster 8088.

The next logical step was a true 32-bit CPU on a chip, the 80386, brought out in 1985. Like the 80286, this one was more-or-less compatible with everything back to the 8080. Being backward compatible was a boon to people for whom running old software was important, but a nuisance to people who would have preferred a simple, clean, modern architecture unencumbered by the mistakes and technology of the past.

Four years later the 80486 came out. It was essentially a faster version of the 80386 that also had a floating-point unit and 8K of cache memory on chip. **Cache memory** is used to hold the most commonly used memory words inside or close to the CPU, to avoid (slow) accesses to main memory. The 80386 also had built-in multiprocessor support, to allow manufacturers to build systems containing multiple CPUs.

At this point, Intel found out the hard way (by losing a trademark infringement lawsuit) that numbers (like 80486) cannot be trademarked, so the next generation got a name: Pentium (from the Greek word for five, πεντε). Unlike the 80486, which had one internal pipeline, the Pentium had two of them, which helped make it twice as fast (we will discuss pipelines in detail in Chap. 2).

When the next generation appeared, people who were hoping for the Sexium (*sex* is Latin for six) were disappointed. The name Pentium was now so well known that the marketing people wanted to keep it, and the new chip was called the Pentium Pro. Despite the small name change from its predecessor, this processor represented a major break with the past. Instead of having two or more pipelines, the Pentium Pro had a very different internal organization and could execute up to five instructions at a time.

Another innovation found in the Pentium Pro was a two-level cache memory. The processor chip itself had 8 KB of memory to hold commonly-used instructions and 8 KB of memory to hold commonly-used data. In the same cavity within the Pentium Pro package (but not on the chip itself) was a second cache memory of 256 KB.

The next new Intel processor was the Pentium II, essentially a Pentium Pro with special multimedia extensions (called **MMX**) added. These instructions were intended to speed up computations required to process audio and video, making the addition of special multimedia coprocessors unnecessary. These instructions were also available in later Pentiums, but not in the Pentium Pro, so the Pentium II combined the strengths of the Pentium Pro with multimedia.

In early 1998, Intel introduced a new product line called the **Celeron**, which was basically a low-price, low-performance version of the Pentium II intended for

low-end PCs. Since the Celeron has the same architecture as the Pentium II, we will not discuss it further in this book. In June 1998, Intel introduced a special version of the Pentium II for the upper end of the market. This processor, called the **Xeon**, had a larger cache, a faster bus, and better multiprocessor support, but was otherwise a normal Pentium II, so we will not discuss it separately either. The Intel family is shown in Fig. 1-1.

Chip	Date	MHz	Transistors	Memory	Notes
4004	4/1971	0.108	2,300	640	First microprocessor on a chip
8008	4/1972	0.108	3,500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6,000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5-10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5-8	29,000	1 MB	Used in IBM PC
80286	2/1982	8-12	134,000	16 MB	Memory protection present
80386	10/1985	16-33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25-100	1.2M	4 GB	Built-in 8K cache memory
Pentium	3/1993	60-233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150-200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233-400	7.5M	4 GB	Pentium Pro plus MMX

Figure 1-1. The Intel CPU family. Clock speeds are measured in MHz (megahertz) where 1 MHz is 1 million cycles/sec.

All the Intel chips are backward compatible with their predecessors back as far as the 8086. In other words, a Pentium II can run 8086 programs without modification. This compatibility has always been a design requirement for Intel, to allow users to maintain their existing investment in software. Of course, the Pentium II is 250 times more complex than the 8086, so it can do quite a few things that the 8086 could not do. These piecemeal extensions have resulted in an architecture that is not as elegant as it might have been had someone given the Pentium II architects 7.5 million transistors and instructions to start all over again.

It is interesting to note that although Moore's law was long associated with the number of bits in a memory, it applies equally well to CPU chips. By plotting the transistor counts given in Fig. 1-1 against the date of introduction of each chip on a semilog scale, we see that Moore's law holds here too. This graph is given in Fig. 1-2.

1.4.2 Introduction to the UltraSPARC II

In the 1970s, UNIX was popular at universities, but no personal computers ran UNIX, so UNIX-lovers had to use (often overloaded) timeshared minicomputers such as the PDP-11 and VAX. In 1981, a German Stanford graduate student,

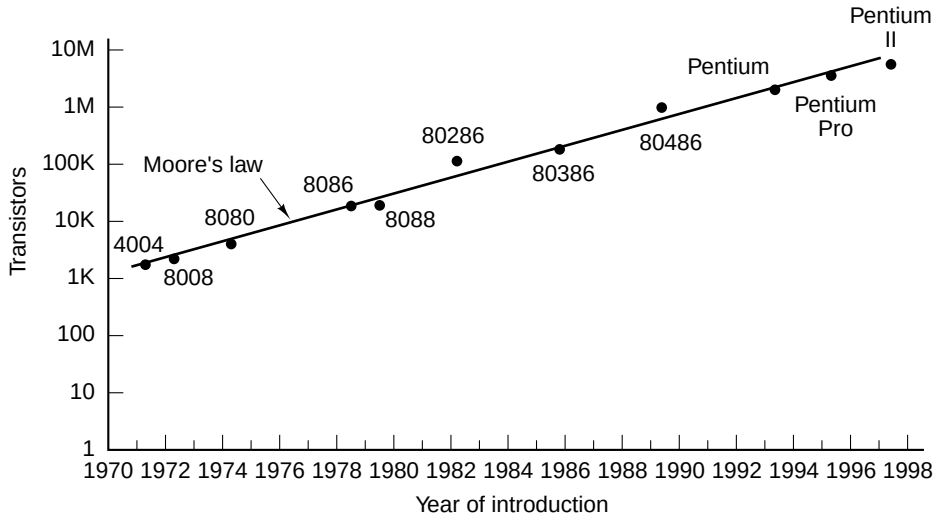


Figure 1-2. Moore's law for CPU chips.

Andy Bechtolsheim, who was frustrated at having to go to the computer center to use UNIX, decided to solve this problem by building himself a personal UNIX workstation out of off-the-shelf parts. He called it the SUN-1 (Stanford University Network).

Bechtolsheim soon attracted the attention of Vinod Khosla, a 27-year-old Indian who had a burning desire to retire as a millionaire by age 30. Khosla convinced Bechtolsheim to form a company to build and sell Sun workstations. Khosla then hired Scott McNealy, another Stanford graduate student, to head manufacturing. To write the software, they hired Bill Joy, the principle architect of Berkeley UNIX. The four of them founded Sun Microsystems in 1982.

Sun's first product, the Sun-1, which was powered by a Motorola 68020 CPU, was an instant success, as were the follow-up Sun-2 and Sun-3 machines, which also used Motorola CPUs. Unlike other personal computers of the day, these machines were far more powerful (hence the designation ''workstation'') and were designed from the start to be run on a network. Each Sun workstation came equipped with an Ethernet connection and with TCP/IP software for connecting to the ARPANET, the forerunner of the Internet.

By 1987, Sun, now selling half a billion dollars a year worth of systems, decided to design its own CPU, basing it upon a revolutionary new design from the University of California at Berkeley (the RISC II). This CPU, called the **SPARC** (**Scalable Processor ARChitecture**), formed the basis of the Sun-4 workstation. Within a short time, all of Sun's products used the SPARC CPU.

Unlike many other computer companies, Sun decided not to manufacture the SPARC CPU chip itself. Instead, it licensed several different semiconductor manufacturers to produce them, hoping that competition among them would drive

performance up and prices down. These vendors produced a number of different chips, based on different technologies, running at different clock speeds, and with various prices. These chips included the MicroSPARC, HyperSPARC, SuperSPARC, and TurboSPARC. Although these CPUs differed in minor ways, all were binary compatible and ran the same user programs without modification.

Sun always wanted SPARC to be an open architecture, with many suppliers of parts and systems, in order to build an industry that could compete in a personal computer world already dominated by Intel-based CPUs. To gain the trust of companies that were interested in the SPARC but did not want to invest in a product controlled by a competitor, Sun created an industry consortium, SPARC International, to manage the development of future versions of the SPARC architecture. Thus it is important to distinguish between the SPARC architecture, which is a specification of the instruction set and other programmer-visible features, and a particular implementation of it. In this book we will study both the generic SPARC architecture, and, when discussing CPU chips in Chaps. 3 and 4, a specific SPARC chip used in Sun workstations.

The initial SPARC was a full 32-bit machine, running at 36 MHz. The CPU, called the **IU** (**Integer Unit**) was lean and mean, with only three major instruction formats and only 55 instructions in all. In addition, a floating-point unit added another 14 instructions. This history can be contrasted to the Intel line, which started out with 8- and 16-bit chips (8088, 8086, 80286) and finally became a 32-bit chip with the 80386.

The SPARC's first break with the past occurred in 1995, with the development of Version 9 of the SPARC architecture, a full 64-bit architecture, with 64-bit addresses and 64-bit registers. The first Sun workstation to implement the V9 (Version 9) architecture was the **UltraSPARC I**, introduced in 1995 (Tremblay and O'Connor, 1996). Despite its being a 64-bit machine, it was also fully binary compatible with the existing 32-bit SPARCs.

The UltraSPARC was intended to break new ground. Whereas previous machines were designed for handling alphanumeric data and running programs like word processors and spreadsheets, the UltraSPARC was designed from the beginning to handle images, audio, video, and multimedia in general. Among other innovations besides the 64-bit architecture were 23 new instructions, including some for packing and unpacking pixels from 64-bit words, scaling and rotating images, block moves, and performing real-time video compression and decompression. These instructions, called **VIS** (**Visual Instruction Set**) were aimed at providing general multimedia capability, analogous to Intel's MMX instructions.

The UltraSPARC was aimed at high-end applications, such as large multiprocessor Web servers with dozens of CPUs and physical memories of up to 2 TB [1 TB (terabyte) = 10^{12} bytes]. However, smaller versions can be used in notebook computers as well.

The successors to the UltraSPARC I were the UltraSPARC II and UltraSPARC III. These models differ primarily in clock speed, but some new features

were added in each iteration as well. In this book, when we discuss the SPARC architecture, we will use the 64-bit V9 UltraSPARC II as our example.

1.4.3 Introduction to the picoJava II

The C programming language was invented by Dennis Ritchie of Bell Labs for use in the UNIX operating system. Due to its economical design and the popularity of UNIX, C soon became the dominant programming language in the world for systems programming. Some years later, Bjarne Stroustrup, also of Bell Labs, added ideas from the world of object-oriented programming to C to produce C++, which also became very popular.

In the mid 1990s, researchers at Sun Microsystems were investigating ways of allowing users to fetch binary programs over the Internet and run them as part of World Wide Web pages. They liked C++, except that it was not secure. In other words, a newly fetched C++ binary program could easily spy on and otherwise interfere with the machine that had just acquired it. Their solution to this was to invent a new programming language, Java, inspired by C++, but without the latter's security problems. Java is a type-safe object-oriented language which is increasingly used for many applications. As it is a popular and elegant language, we will use it in this book for programming examples.

Since Java is just a programming language, it is possible to write compilers for it that compile to the Pentium, SPARC, or any other architecture. Such compilers exist. However, Sun's major goals in introducing Java was to make it possible to exchange Java executable programs between computers on the Internet and have the receiver run them without modification. If a Java program compiled on a SPARC were shipped over the Internet to a Pentium, it would not run, defeating the goal of being able to send a binary program anywhere and run it there.

To make binary programs portable across different machines, Sun defined a virtual machine architecture called **JVM** (**Java Virtual Machine**). This machine has a memory consisting of 32-bit words and 226 instructions that the machine can execute. Most of these instructions are simple, but a few are quite complex, requiring multiple memory cycles.

To make Java programs portable, Sun wrote a compiler that compiles Java to JVM. It also wrote a JVM interpreter to execute Java binary programs. This interpreter was written in C and thus can be compiled and executed on any machine with a C compiler, which, in practice, means almost every machine in the world. Consequently, to enable a machine to execute Java binary programs, all the machine's owner needs to do is get the executable binary program for the JVM interpreter for that platform (e.g., Pentium II and Windows 98, SPARC and UNIX etc.) along with certain associated support programs and libraries. In addition, most Internet browsers contain a JVM interpreter inside them, to make it very easy to run **applets**, which are little Java binary programs associated with World Wide Web pages. Many of these applets provide animation and sound.

Interpreting JVM programs (or any other programs, for that matter) is slow. An alternative approach to running an applet or other newly-received JVM program is to first compile it for the machine at hand, and then run the compiled program. This strategy requires having a JVM-to-machine-language compiler inside the browser and being able to activate it on-the-fly as needed. Such compilers, called **JIT** (**Just In Time**) compilers, exist and are commonplace. However, they are large and introduce a delay between arrival of the JVM program and its execution while the JVM program is compiled to machine language.

In addition to software implementations of the JVM machine (JVM interpreters and JIT compilers), Sun and other companies have designed hardware JVM chips. These are CPU designs that directly execute JVM binary programs, without the need for a layer of software interpretation or JIT compilation. The initial architectures, the picoJava-I (O'Connor and Tremblay, 1997) and the picoJava-II (McGhan and O'Connor, 1998), were targeted at the embedded systems market. This market requires powerful, flexible, and especially low-cost chips (under $50, often way under) that are embedded inside smart cards, TV sets, telephones, and other appliances, especially those that need to communicate with the outside world. Sun licensees can manufacture their own chips using the picoJava design, customizing them to some extent by including or removing the floating-point unit, adjusting the size of the caches, etc.

The value of a Java chip for the embedded systems market is that a device can change its functionality while in operation. As an example, consider a business executive with a Java-based cellular telephone who never anticipated the need to read faxes on the telephone's tiny screen and who suddenly needs to do so. By calling the cellular provider, the executive can download a fax viewing applet into the telephone and add this functionality to the device. Performance requirements dictate against having Java applets be interpreted, and the lack of memory in the telephone make JIT compilation impossible. This is a situation where a JVM chip is useful.

Although the picoJava II is not a concrete chip (you cannot go to the store and buy one), it is the basis for a number of chips, such as the Sun microJava 701 CPU and various chips from other Sun licensees. We will use the picoJava II design as one of our running examples throughout the book since it is very different from the Pentium II and UltraSPARC CPUs and is targeted at a very different application area. This CPU is particularly interesting for our purposes because in Chap. 4, we will present a design for implementing a subset of JVM using microprogramming. We will then be able to contrast our microprogrammed design with a true hardware design.

The picoJava II has two optional units: a cache and a floating-point unit, which each chip manufacturer can include or remove, as it wishes. For the sake of simplicity, we will refer to the picoJava II as if it were a chip rather than a chip design. When it matters (only rarely), we will be specific and talk about the Sun microJava 701 chip that implements the picoJava II design. When we do not

mention the microJava 701 specifically, the material still applies to it, and also to all the other Java chips from other vendors based on this design.

By using the Pentium II, UltraSPARC II, and picoJava II as our examples, we can study three different kinds of CPUs. These are, respectively, a traditional CISC architecture implemented with modern superscalar technology, a true RISC architecture implemented with superscalar technology, and a dedicated Java chip for use in embedded systems. These three processors are very different from one another, which gives us the opportunity to explore the design space better and see what kinds of trade-offs can be made for processors aimed at different audiences.

2

COMPUTER SYSTEMS ORGANIZATION

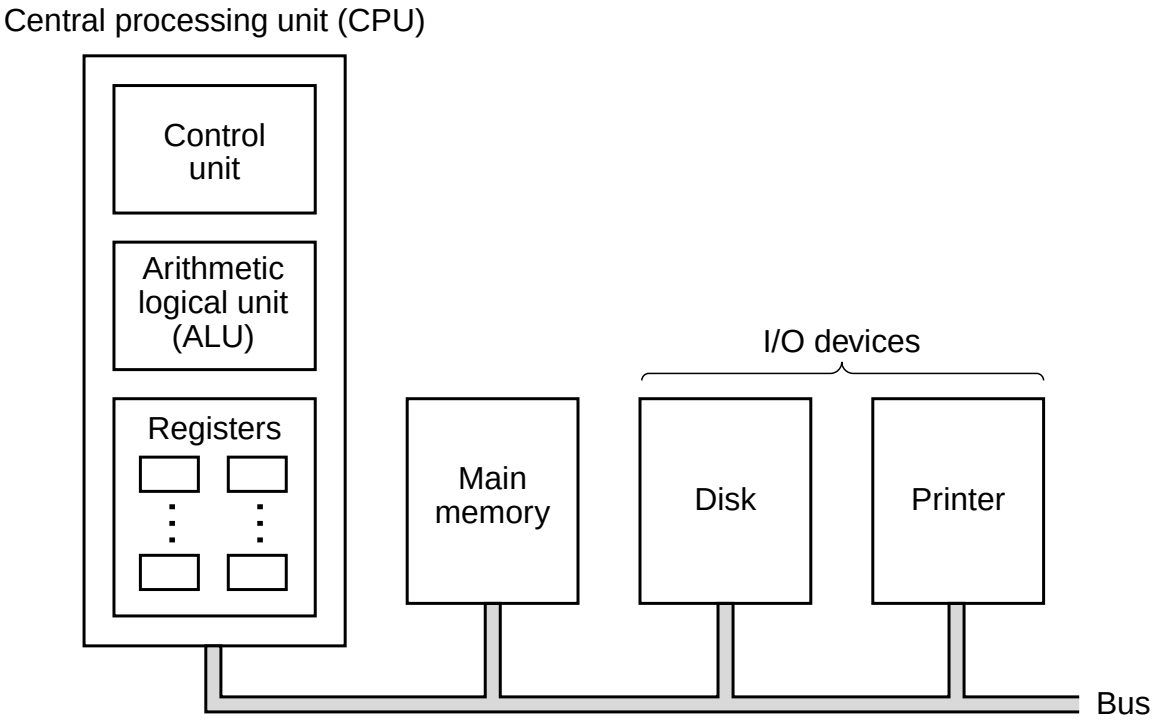


Figure 2-1. The organization of a simple computer with one CPU and two I/O devices.

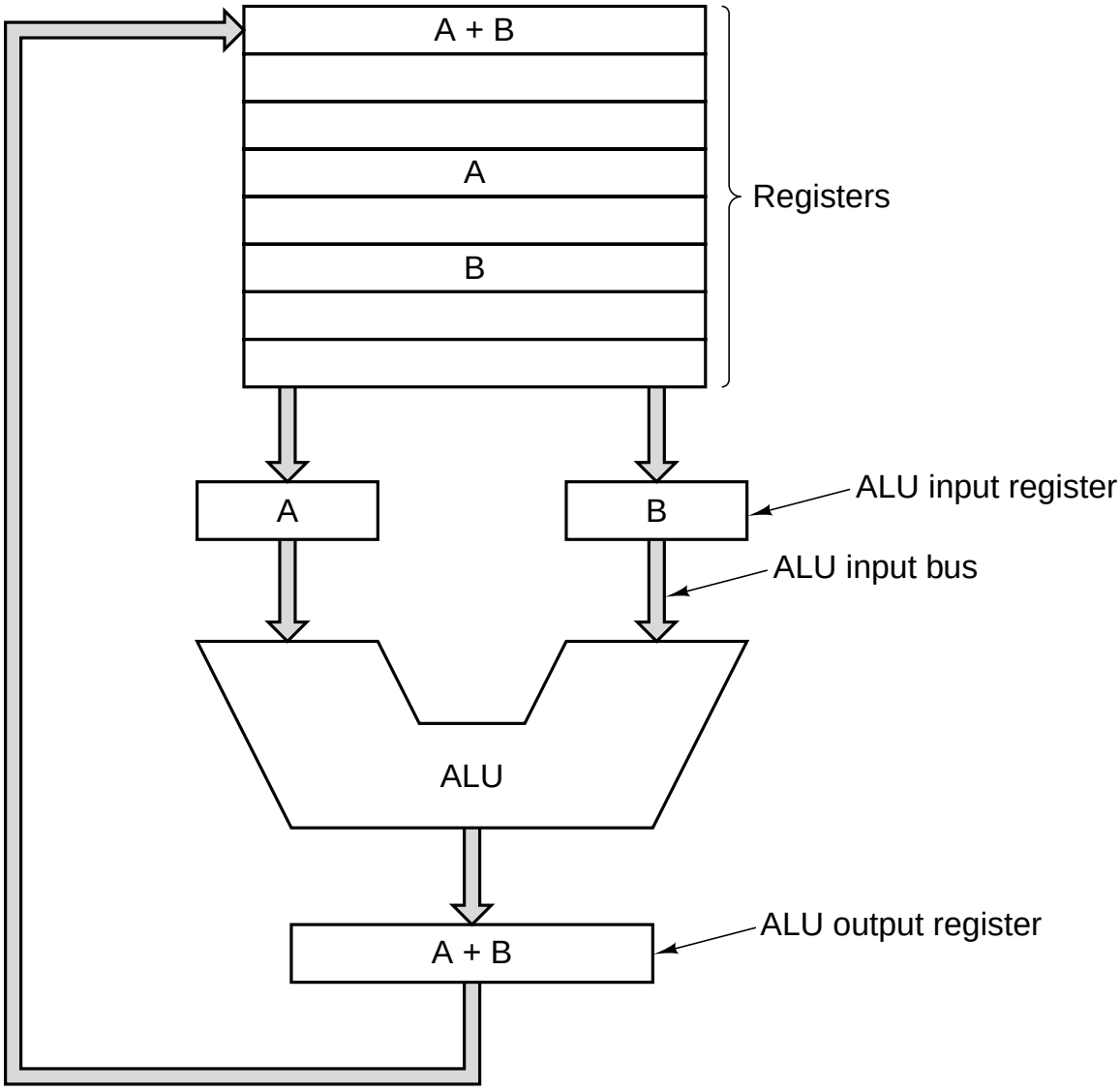


Figure 2-2. The data path of a typical von Neumann machine.

```
public class Interp {
   static int PC;                        // program counter holds address of next instr
   static int AC;                        // the accumulator, a register for doing arithmetic
   static int instr;                     // a holding register for the current instruction
   static int instr_type;                // the instruction type (opcode)
   static int data_loc;                  // the address of the data, or –1 if none
   static int data;                      // holds the current operand
   static boolean run_bit = true;        // a bit that can be turned off to halt the machine

   public static void interpret(int memory[ ], int starting_address) {
      // This procedure interprets programs for a simple machine with instructions having
      // one memory operand. The machine has a register AC (accumulator), used for
      // arithmetic. The ADD instruction adds am integer in memory to the AC, for example
      // The interpreter keeps running until the run bit is turned off by the HALT instruction.
      // The state of a process running on this machine consists of the memory, the
      // program counter, the run bit, and the AC. The input parameters consist of
      // of the memory image and the starting address.

      PC = starting_address;
      while (run_bit) {
         instr = memory[PC];             // fetch next instruction into instr
         PC = PC + 1;                    // increment program counter
         instr_type = get_instr_type(instr);          // determine instruction type
         data_loc = find_data(instr, instr_type);     // locate data (–1 if none)
         if (data_loc >= 0)              // if data_loc is –1, there is no operand
            data = memory[data_loc];     // fetch the data
         execute(instr_type, data);      //execute instruction
      }

   }

   private static int get_instr_type(int addr) { ... }
   private static int find_data(int instr, int type) { ... }
   private static void execute(int type, int data){ ... }
}
```

Figure 2-3. An interpreter for a simple computer (written in Java).

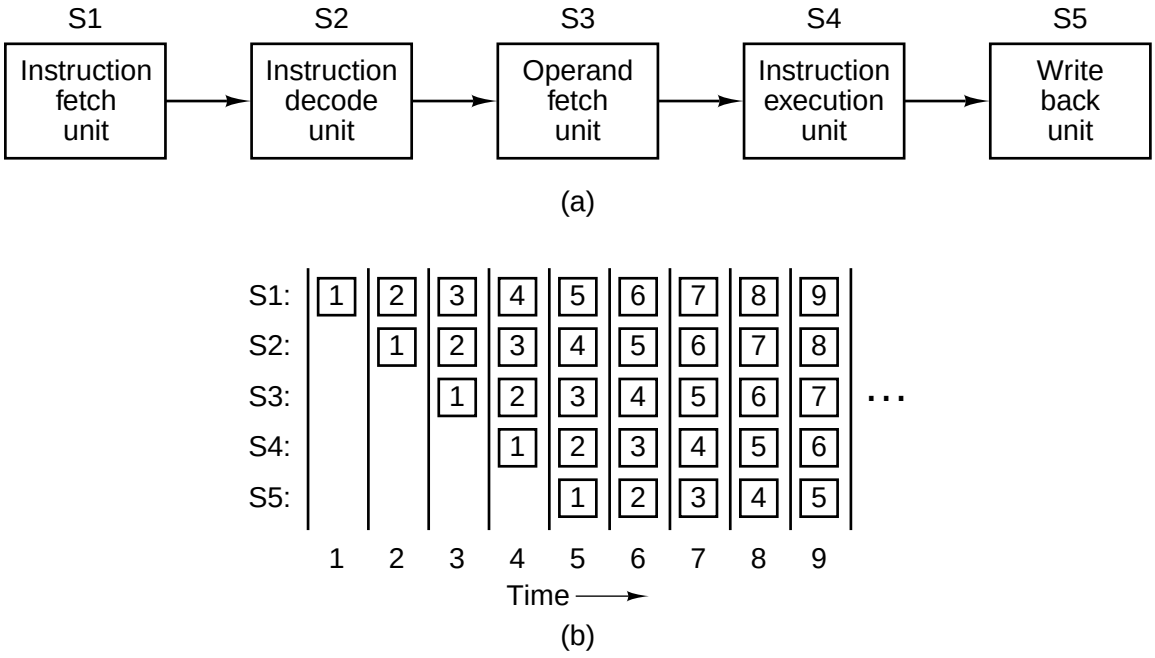


Figure 2-4. (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

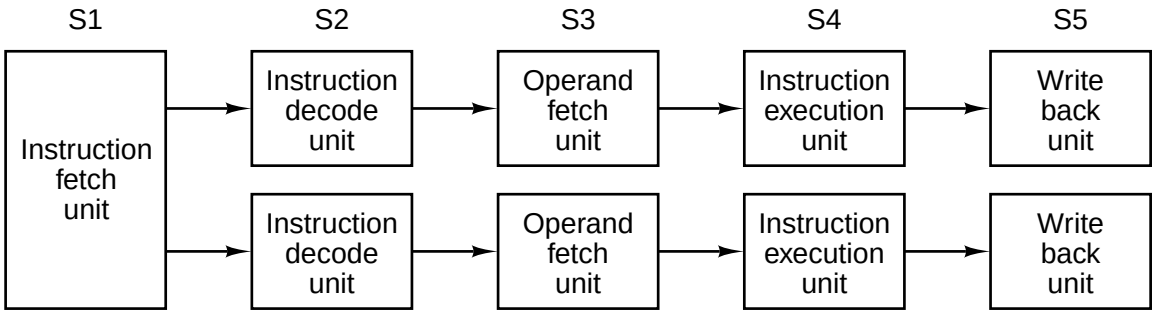


Figure 2-5. (a) Dual five-stage pipelines with a common instruction fetch unit.

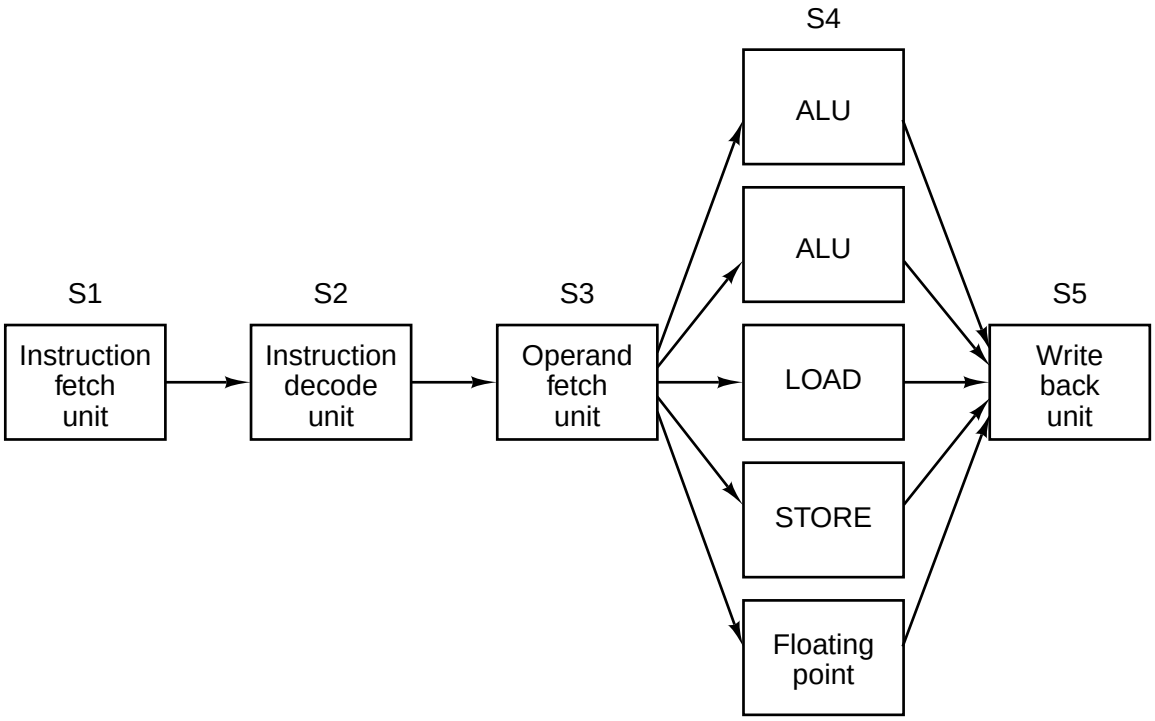


Figure 2-6. A superscalar processor with five functional units.

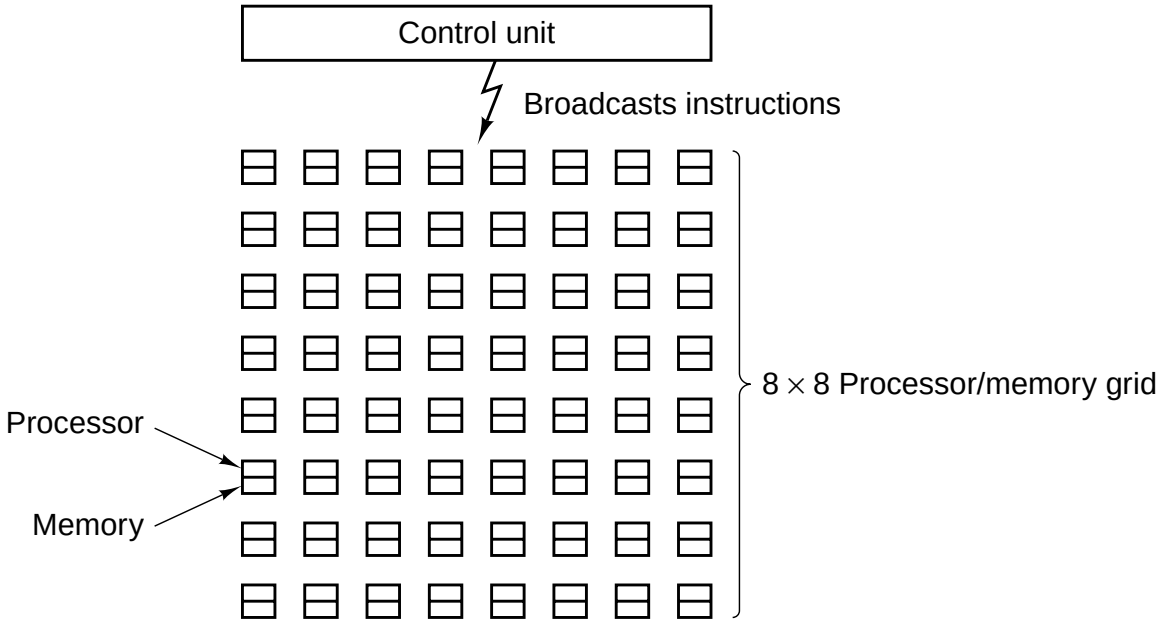


Figure 2-7. An array processor of the ILLIAC IV type.

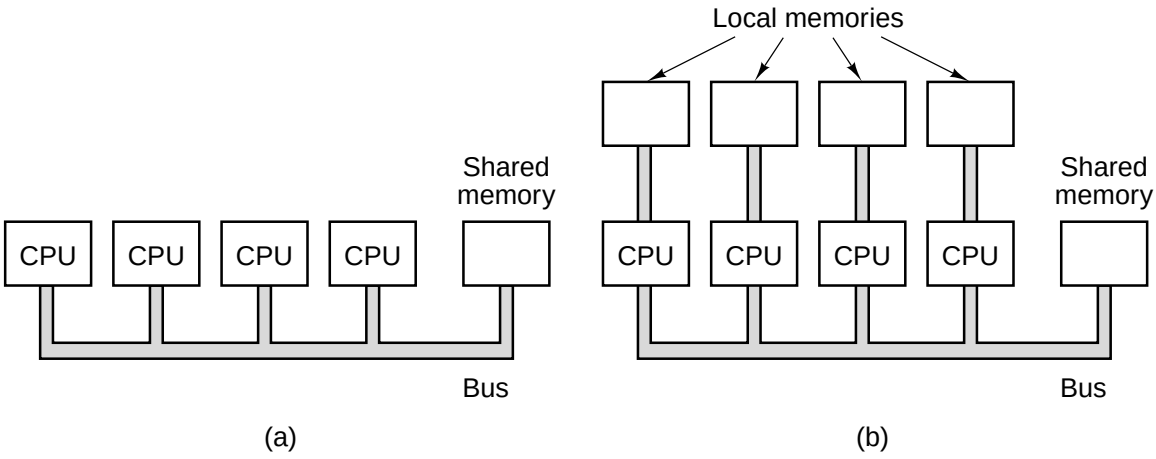


Figure 2-8. (a) A single-bus multiprocessor. (b) A multicomputer with local memories.

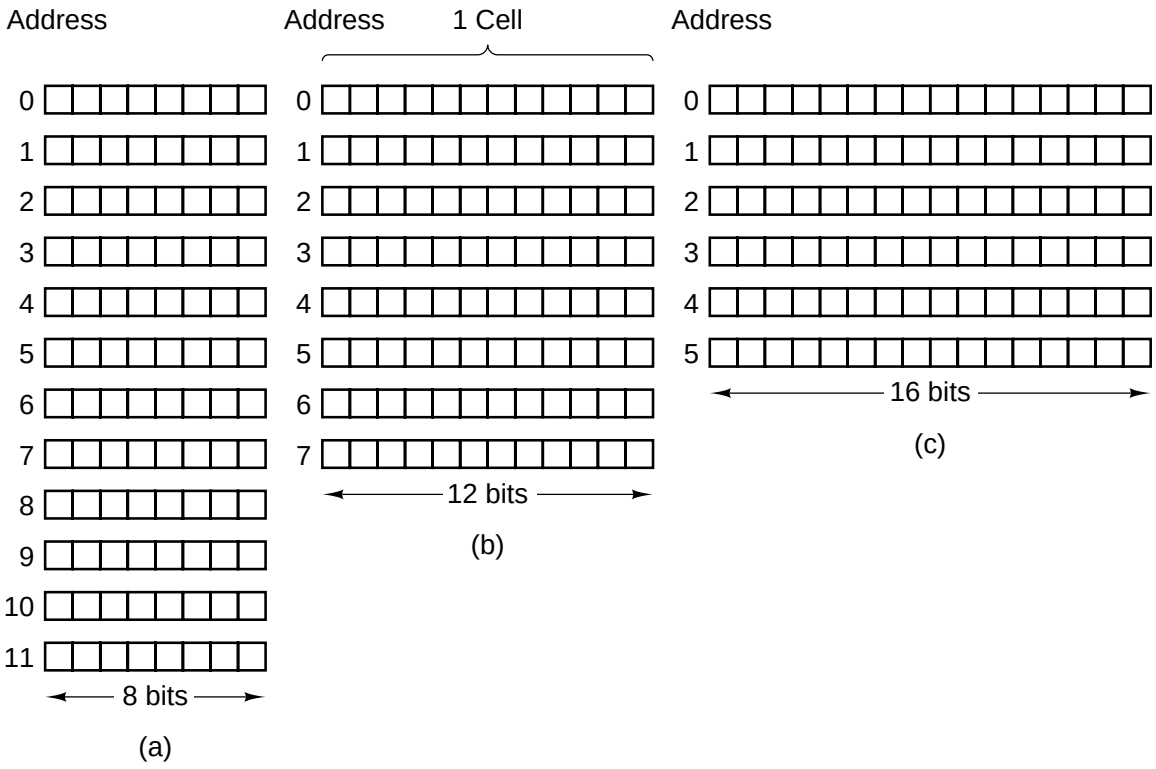


Figure 2-9. Three ways of organizing a 96-bit memory.

Computer	Bits/cell
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

Figure 2-10. Number of bits per cell for some historically interesting commercial computers.

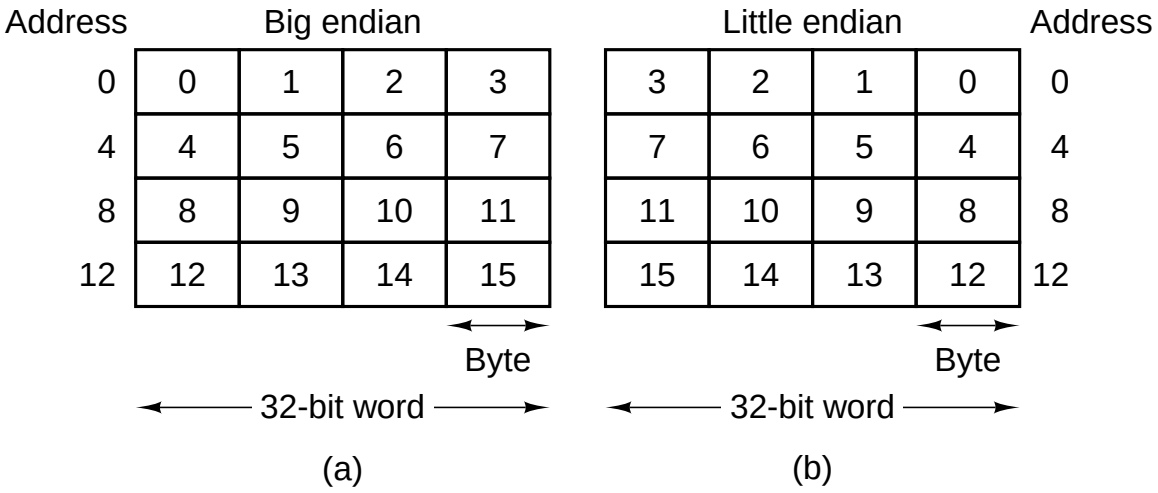


Figure 2-11. (a) Big endian memory. (b) Little endian memory.

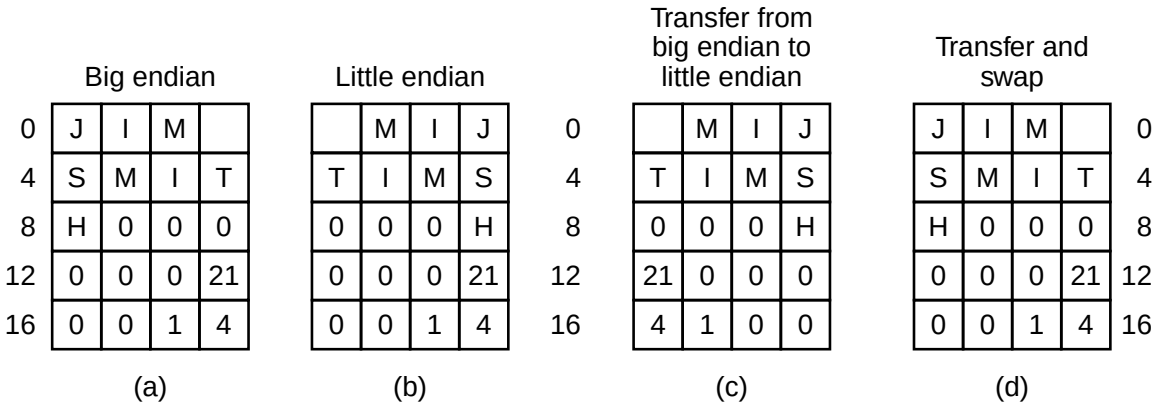


Figure 2-12. (a) A personnel record for a big endian machine. (b) The same record for a little endian machine. (c) The result of transferring the record from a big endian to a little endian. (d) The result of byte-swapping (c).

Word size	Check bits	Total size	Percent overhead
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Figure 2-13. Number of check bits for a code that can correct a single error.

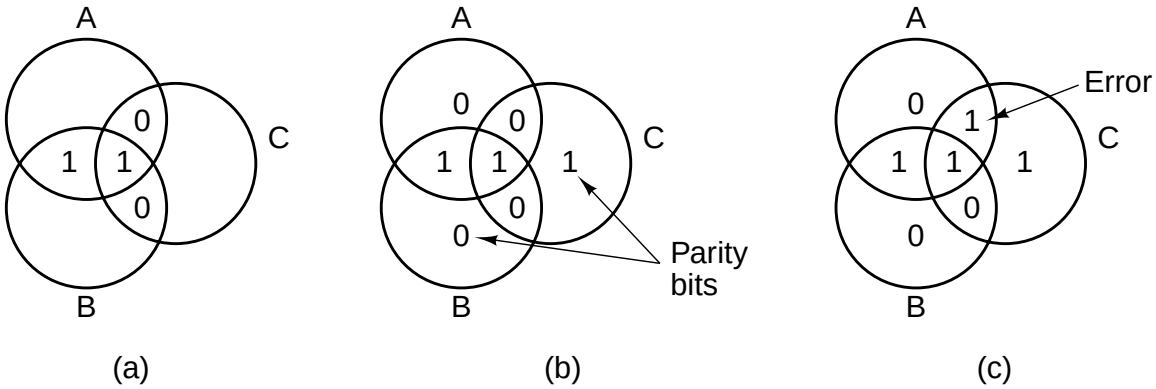


Figure 2-14. (a) Encoding of 1100. (b) Even parity added. (c) Error in *AC*.

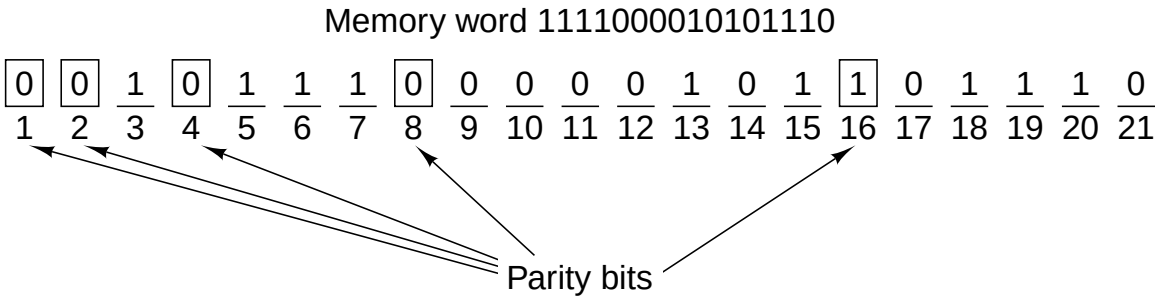


Figure 2-15. Construction of the Hamming code for the memory word 1111000010101110 by adding 5 check bits to the 16 data bits.

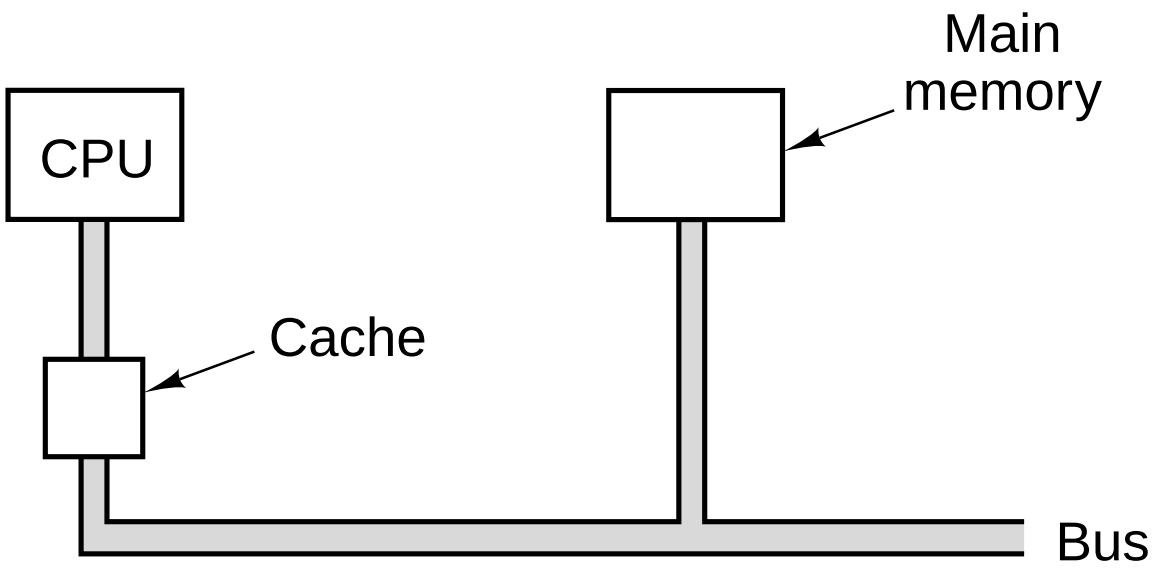


Figure 2-16. The cache is logically between the CPU and main memory. Physically, there are several possible places it could be located.

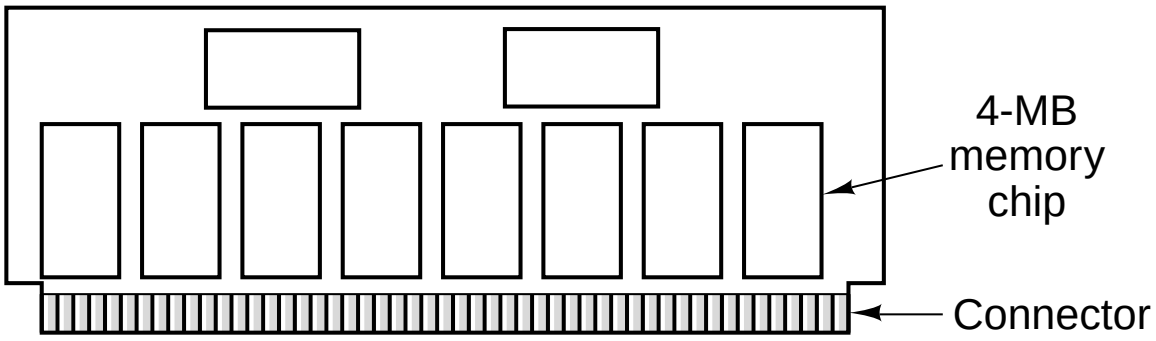


Figure 2-17. A single inline memory module (SIMM) holding 32 MB. Two of the chips control the SIMM.

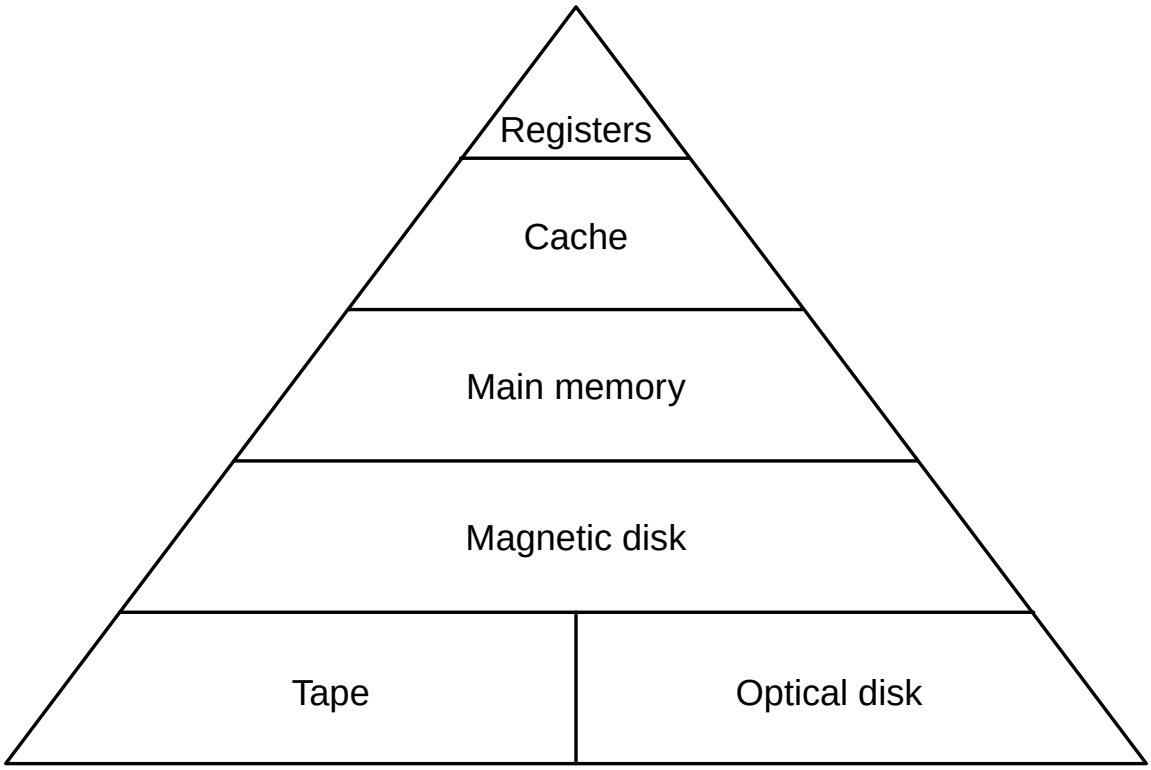


Figure 2-18. A five-level memory hierarchy.

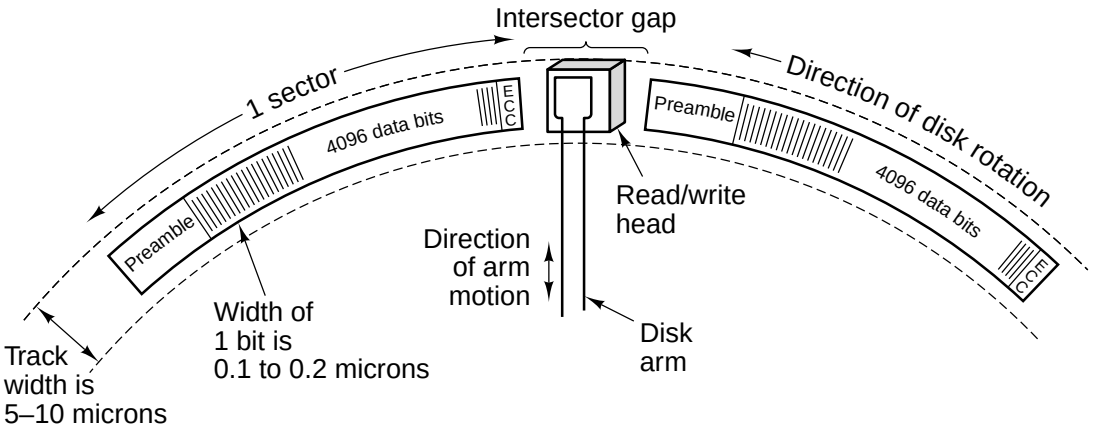


Figure 2-19. A portion of a disk track. Two sectors are illustrated.

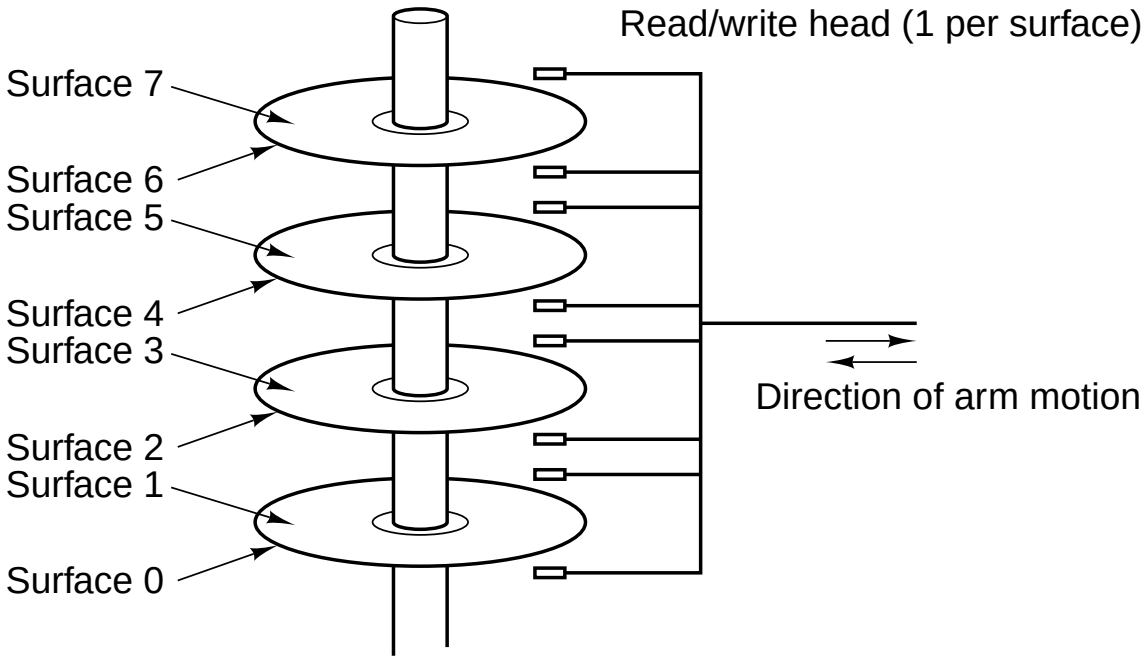


Figure 2-20. A disk with four platters.

Parameters	LD 5.25″	HD 5.25″	LD 3.5″	HD 3.5″
Size (inches)	5.25	5.25	3.5	3.5
Capacity (bytes)	360K	1.2M	720K	1.44M
Tracks	40	80	80	80
Sectors/track	9	15	9	18
Heads	2	2	2	2
Rotations/min	300	360	300	300
Data rate (kbps)	250	500	250	500
Type	Flexible	Flexible	Rigid	Rigid

Figure 2-21. Characteristics of the four kinds of floppy disks.

Name	Data bits	Bus MHz	MB/sec
SCSI-1	8	5	5
SCSI-2	8	5	5
Fast SCSI-2	8	10	10
Fast & wide SCSI-2	16	10	20
Ultra SCSI	16	20	40

Figure 2-22. Some of the possible SCSI parameters.

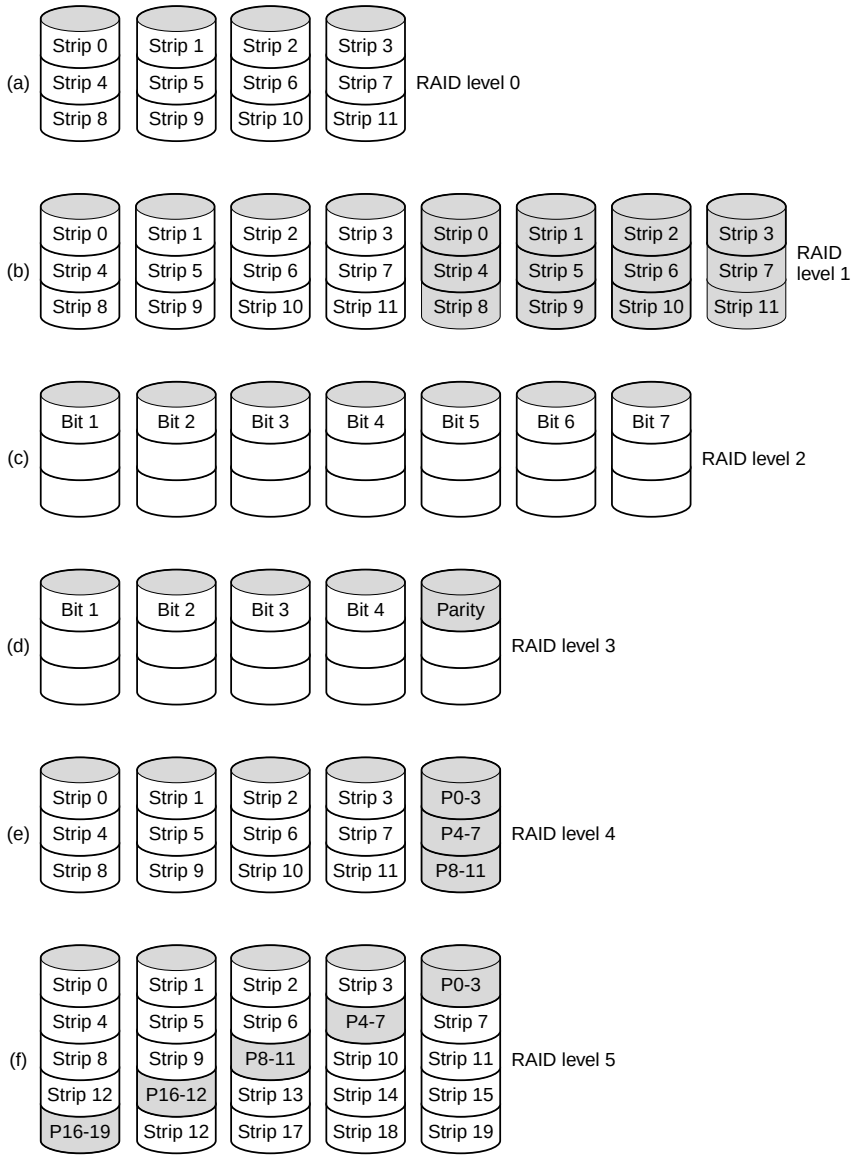


Figure 2-23. RAID levels 0 through 5. Backup and parity drives are shown shaded.

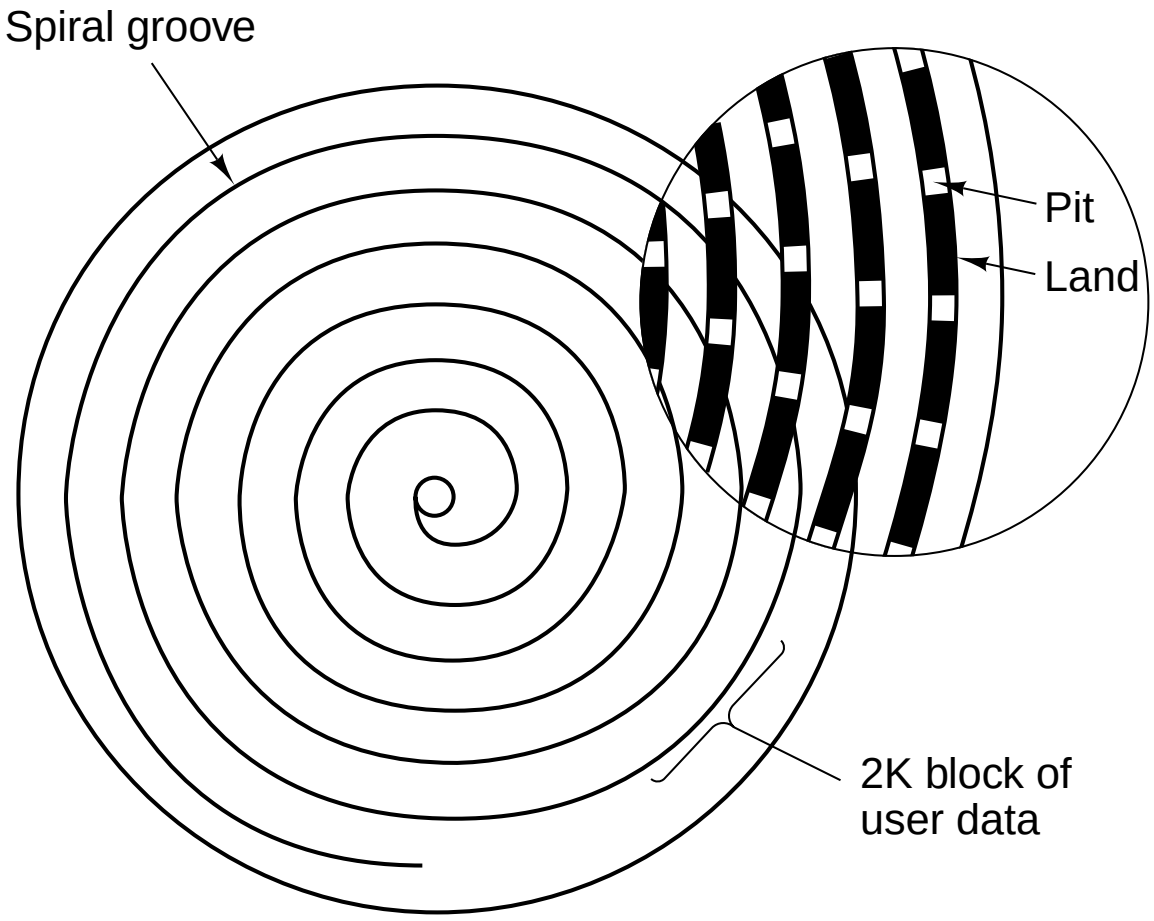


Figure 2-24. Recording structure of a Compact Disc or CD-ROM.

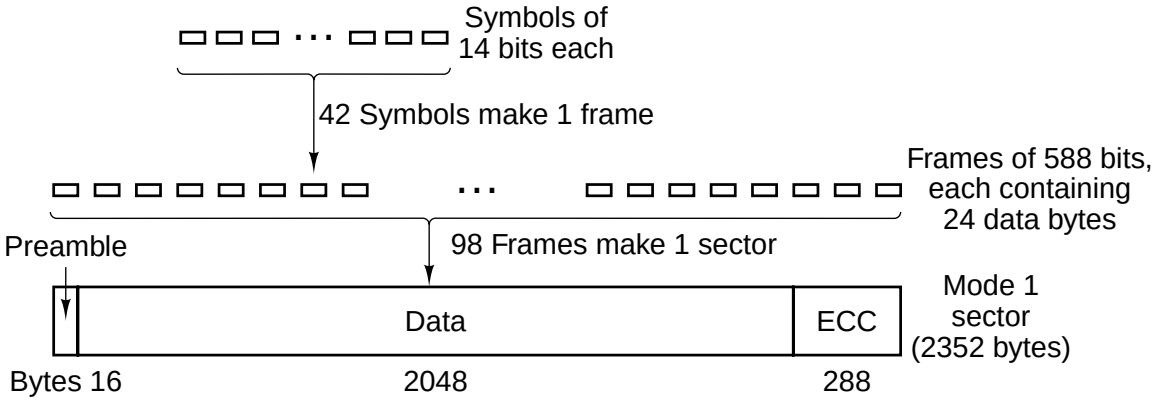


Figure 2-25. Logical data layout on a CD-ROM.

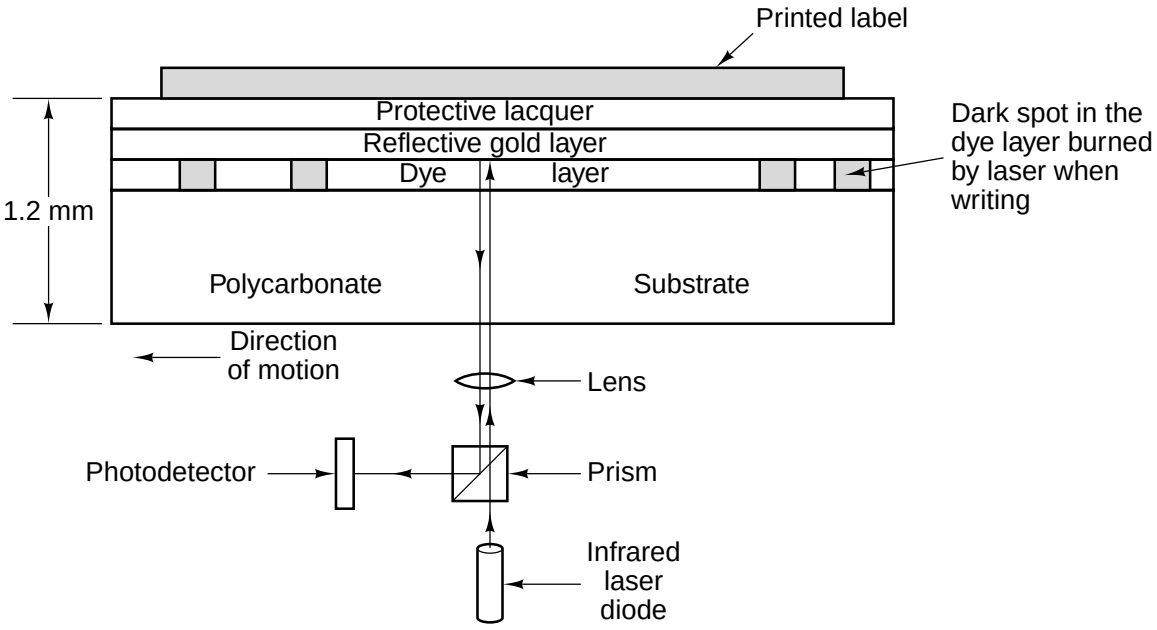


Figure 2-26. Cross section of a CD-R disk and laser (not to scale). A silver CD-ROM has a similar structure, except without the dye layer and with a pitted aluminum layer instead of a gold layer.

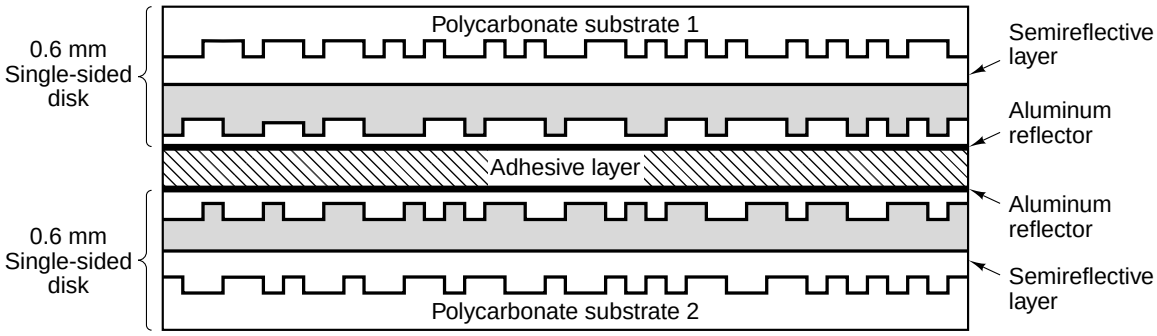


Figure 2-27. A double-sided, dual layer DVD disk.

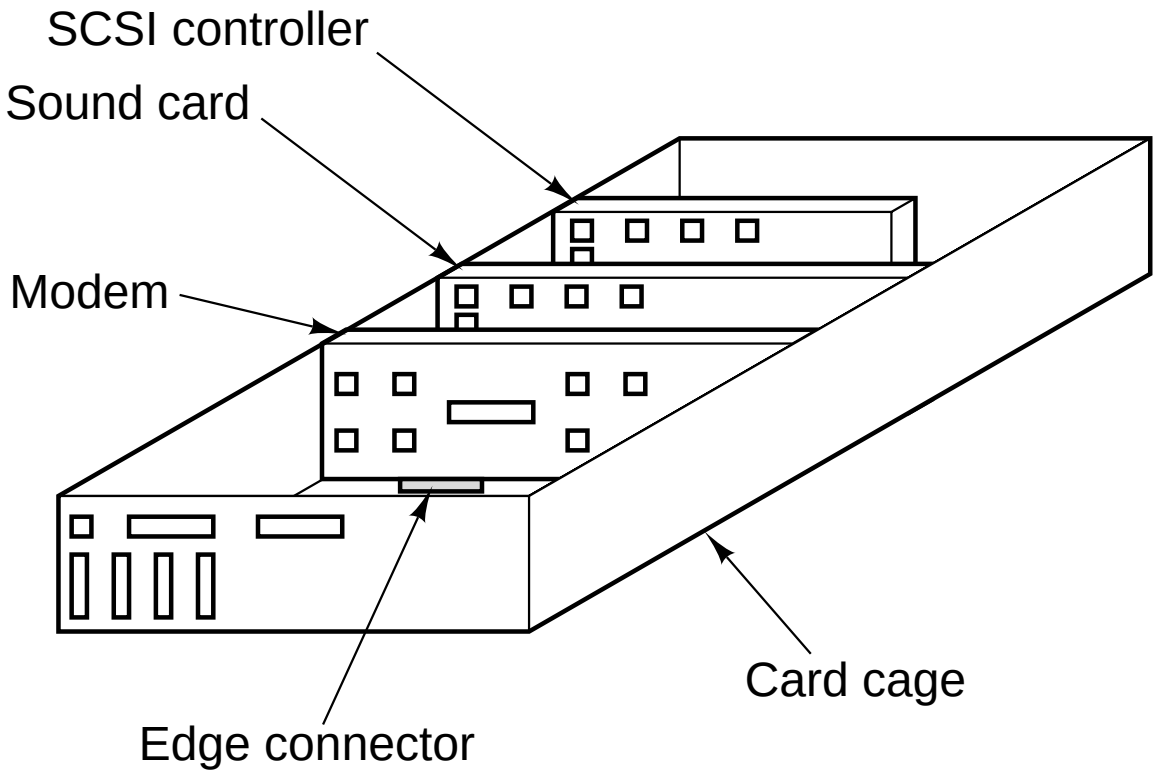


Figure 2-28. Physical structure of a personal computer.

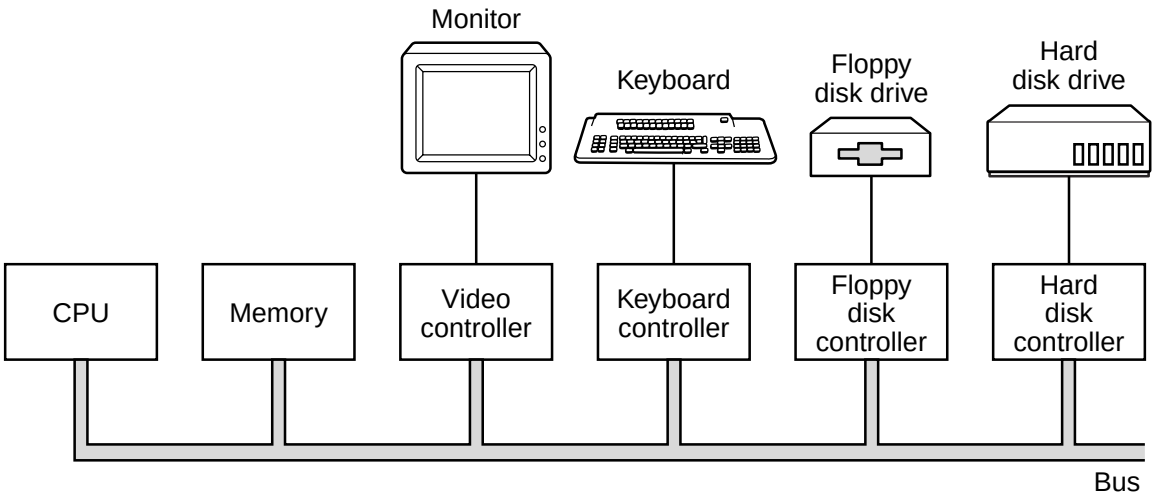


Figure 2-29. Logical structure of a simple personal computer.

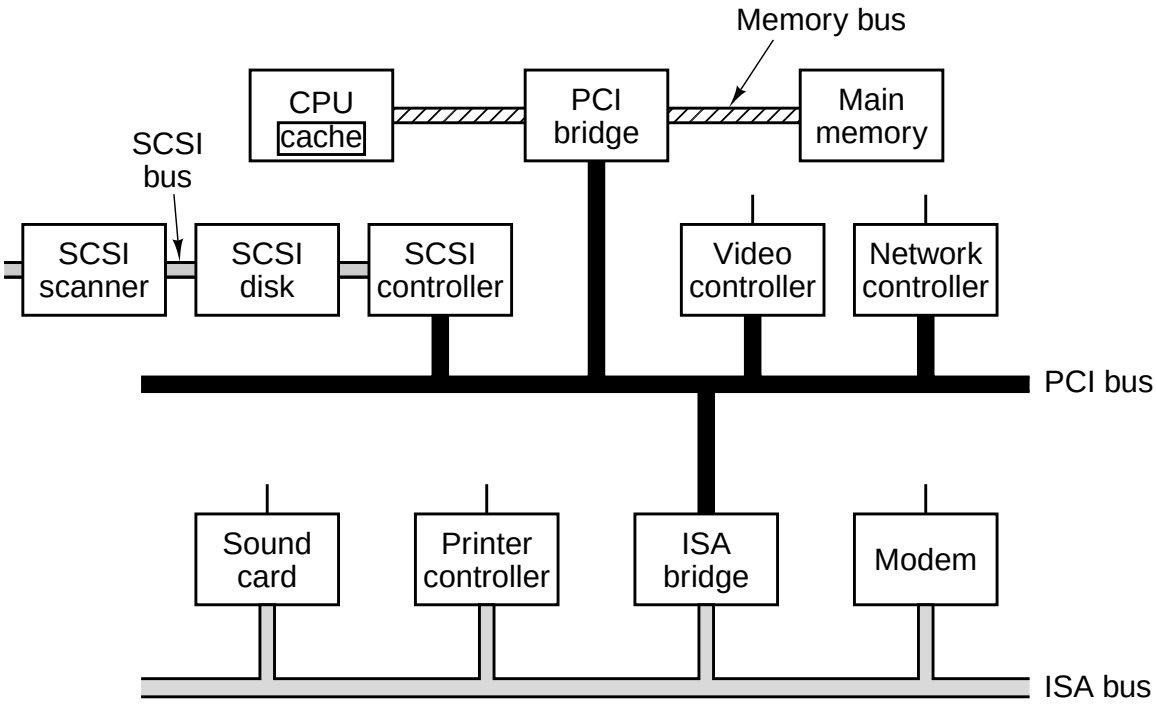


Figure 2-30. A typical modern PC with a PCI bus and an ISA bus. The modem and sound card are ISA devices; the SCSI controller is a PCI device.

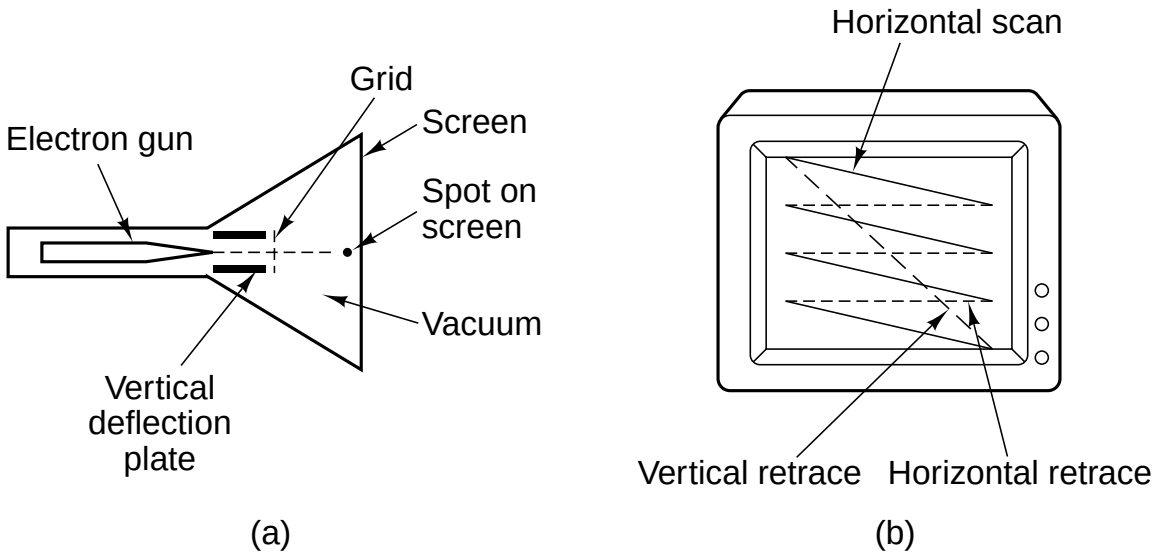


Figure 2-31. (a) Cross section of a CRT. (b) CRT scanning pattern.

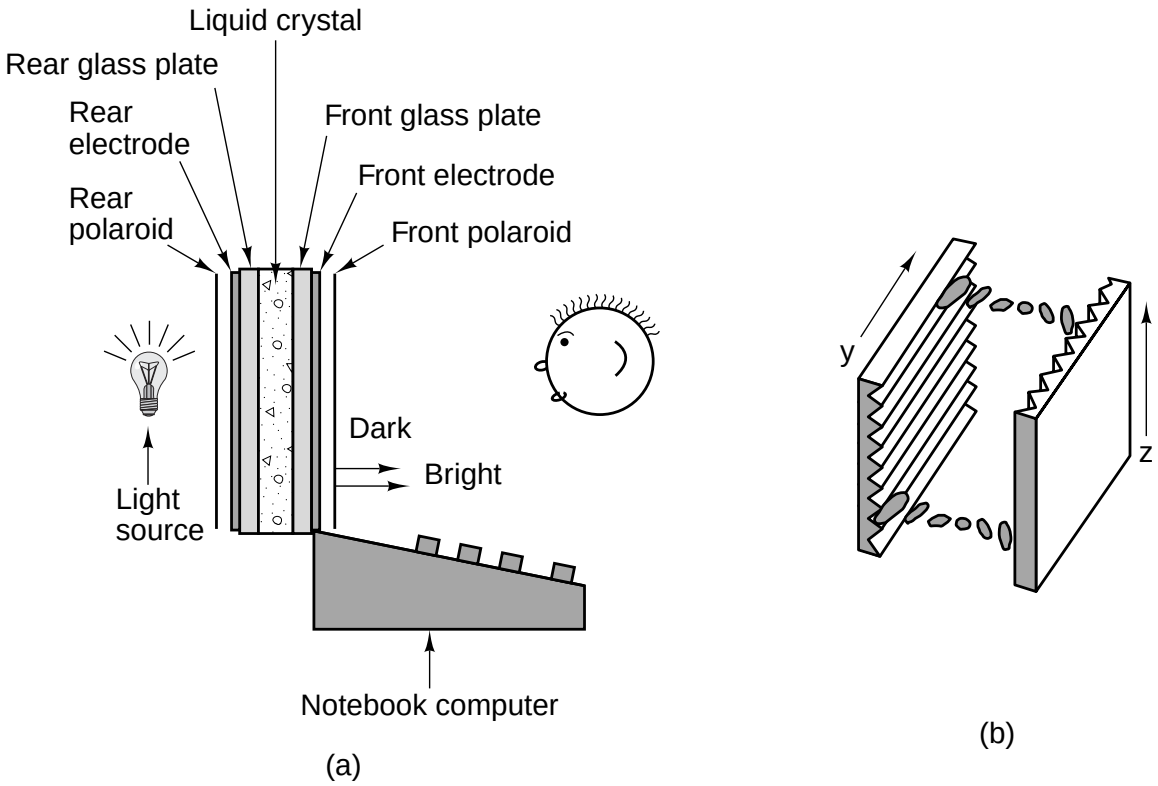


Figure 2-32. (a) The construction of an LCD screen. (b) The grooves on the rear and front plates are perpendicular to one another.

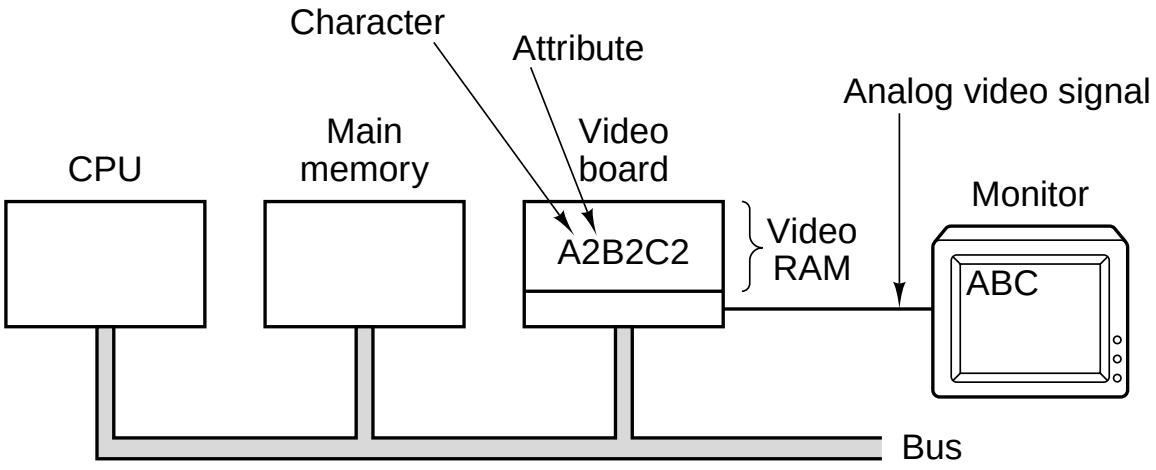


Figure 2-33. Terminal output on a personal computer.

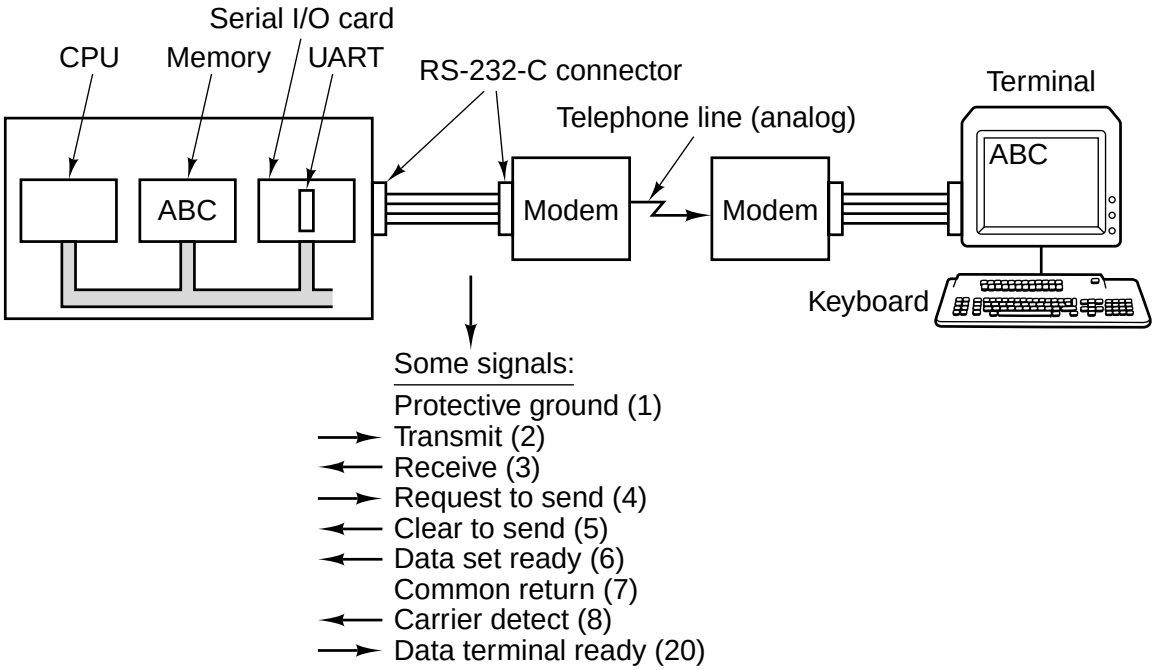


Figure 2-34. Connection of an RS-232-C terminal to a computer. The numbers in parentheses in the list of signals are the pin numbers.

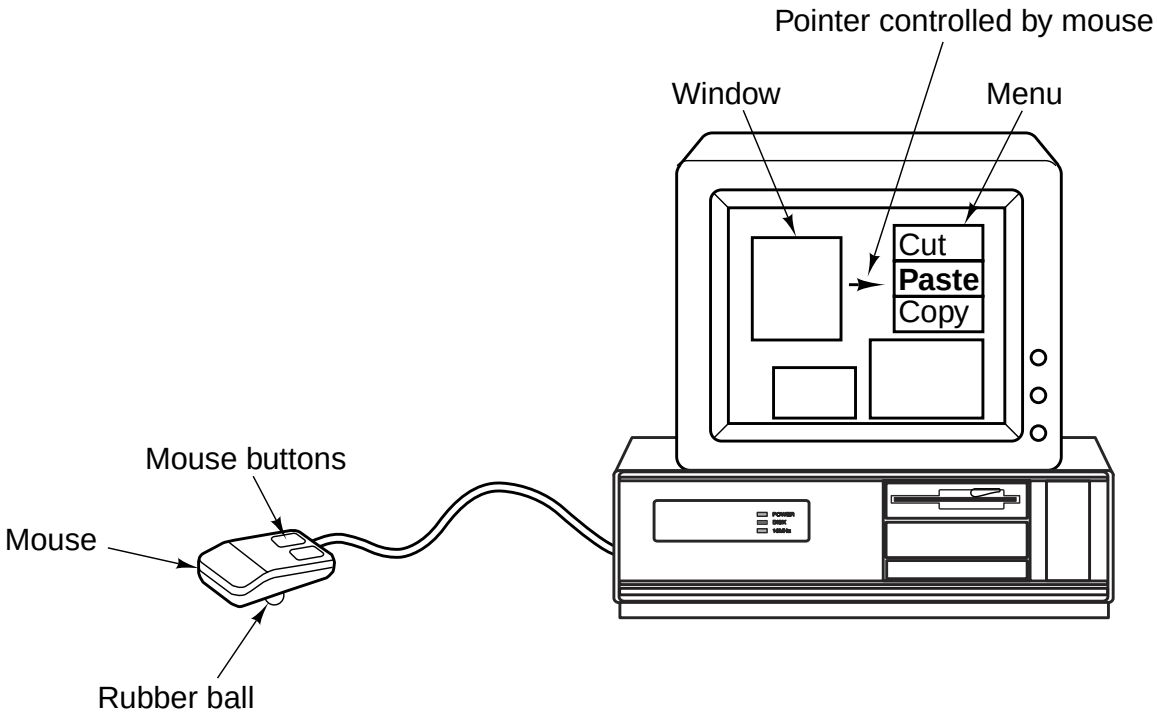


Figure 2-35. A mouse being used to point to menu items.

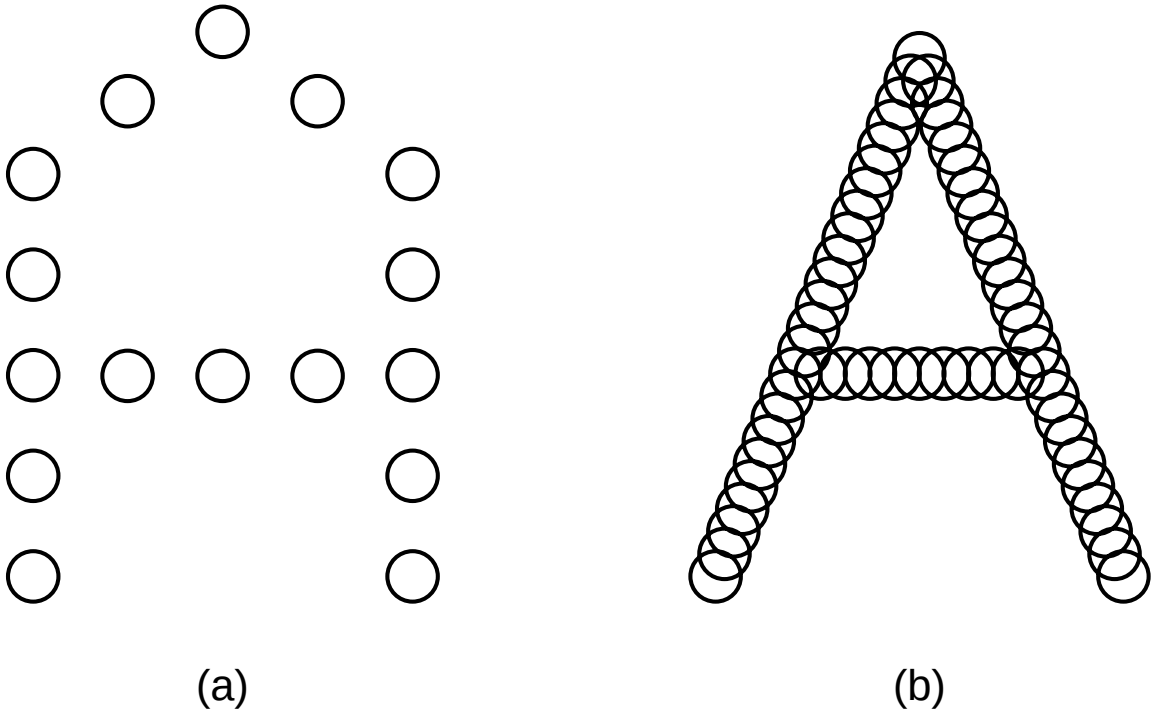


Figure 2-36. (a) The letter ‘‘A’’ on a 5×7 matrix. (b) The letter ‘‘A’’ printed with 24 overlapping needles.

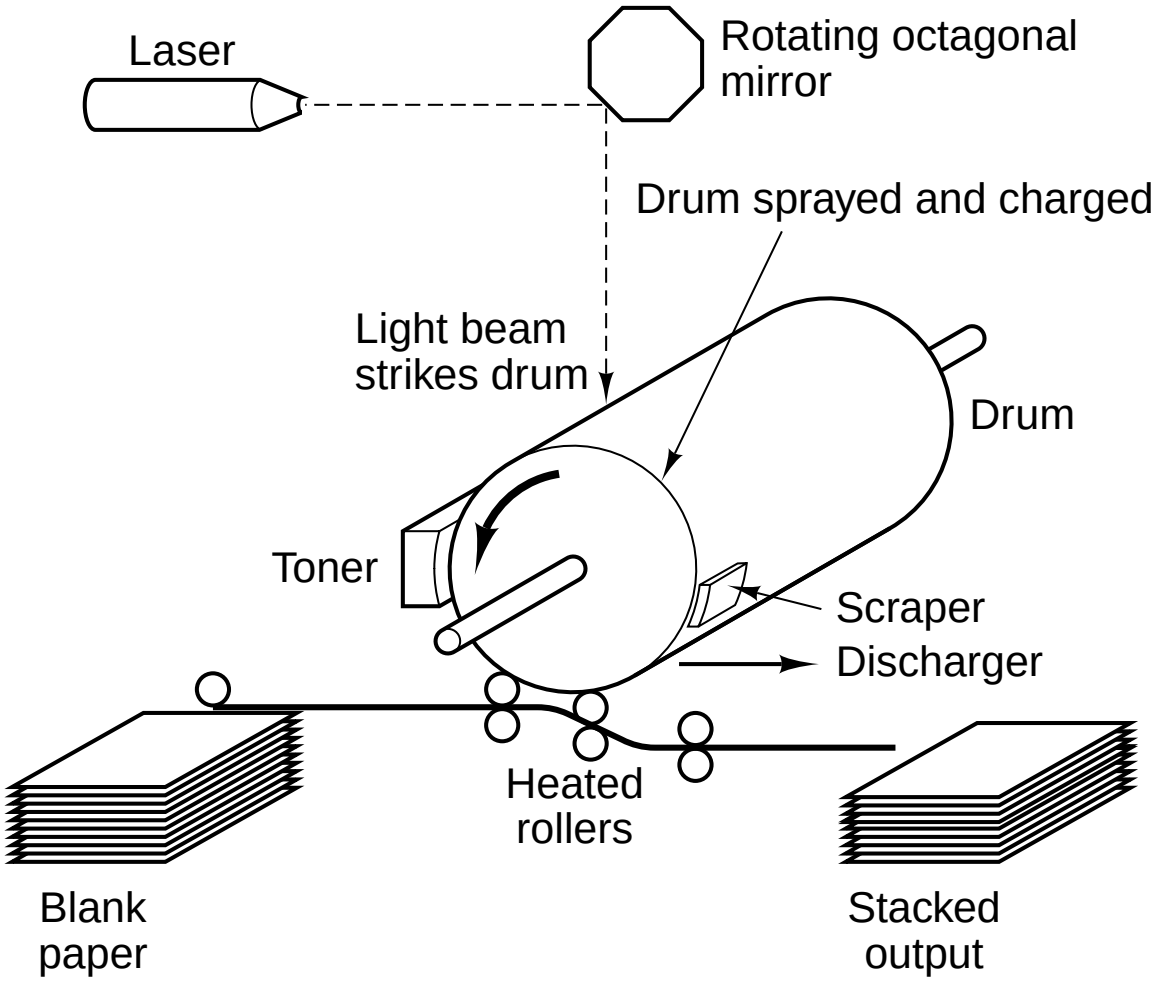


Figure 2-37. Operation of a laser printer.

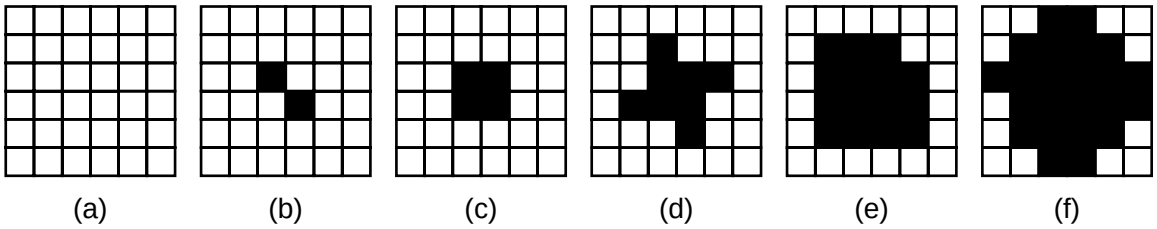


Figure 2-38. Halftone dots for various gray scale ranges. (a) 0–6. (b) 14–20. (c) 28–34. (d) 56–62. (e) 105–111. (f) 161–167.

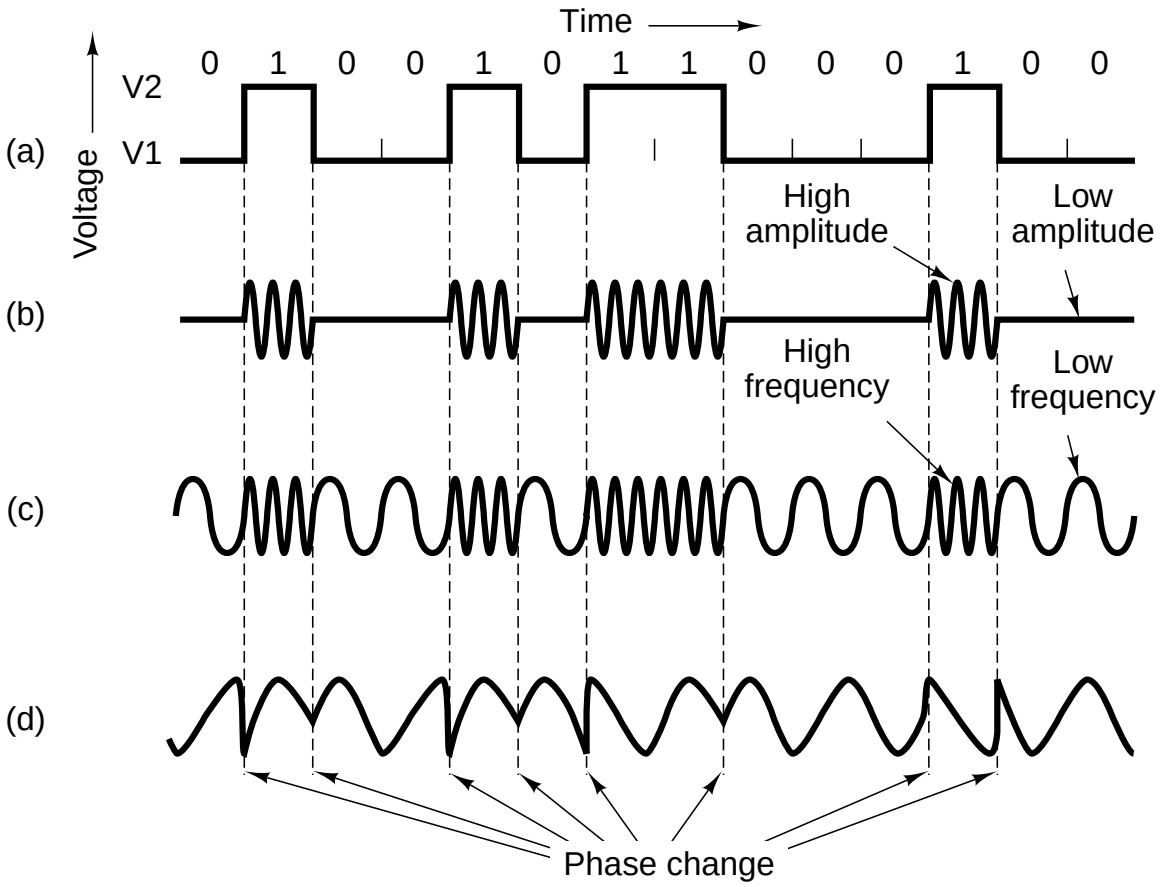


Figure 2-39. Transmission of the binary number 01001011000100 over a telephone line bit by bit. (a) Two-level signal. (b) Amplitude modulation. (c) Frequency modulation. (d) Phase modulation.

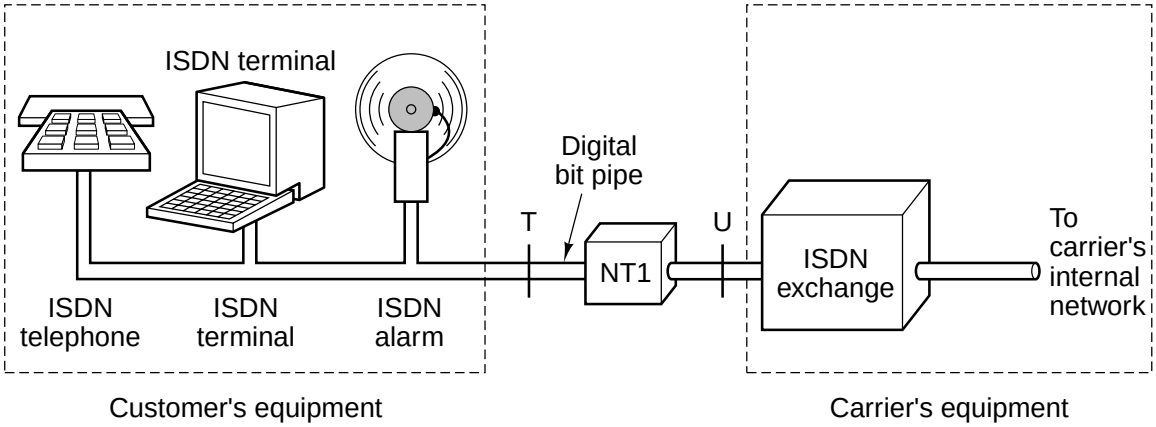


Figure 2-40. ISDN for home use.

2.4 INPUT/OUTPUT

As we mentioned at the start of this chapter, a computer system has three major components: the CPU, the memories (primary and secondary), and the **I/O** (**Input/Output**) equipment such as printers, scanners, and modems. So far we have looked at the CPU and the memories. Now it is time to examine the I/O equipment and how it is connected to the rest of the system.

2.4.1 Buses

Physically, most personal computers and workstations have a structure similar to the one shown in Fig. 2-1. The usual arrangement is a metal box with a large printed circuit board at the bottom, called the **motherboard** (parentboard, for the politically correct). The motherboard contains the CPU chip, some slots into which DIMM modules can be clicked, and various support chips. It also contains a bus etched along its length, and sockets into which the edge connectors of I/O boards can be inserted. Sometimes there are two buses, a high-speed one (for modern I/O boards) and a low-speed one (for older I/O boards).

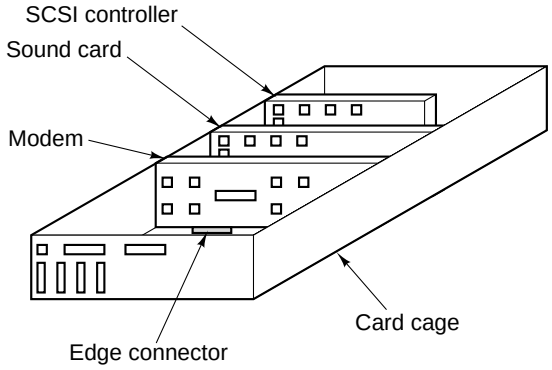


Figure 2-1. Physical structure of a personal computer.

The logical structure of a simple low-end personal computer is shown in Fig. 2-2. This one has a single bus used to connect the CPU, memory, and I/O devices; most systems have two or more buses. Each I/O device consists of two parts: one containing most of the electronics, called the **controller**, and one containing the I/O device itself, such as a disk drive. The controller is usually contained on a board plugged into a free slot, except for those controllers that are not optional (such as the keyboard), which are sometimes located on the motherboard. Even though the display (monitor) is not an option, the video controller is sometimes located on a plug-in board to allow the user to choose between boards with or without graphics accelerators, extra memory, and so on. The controller connects to its device by a cable attached to a connector on the back of the box.

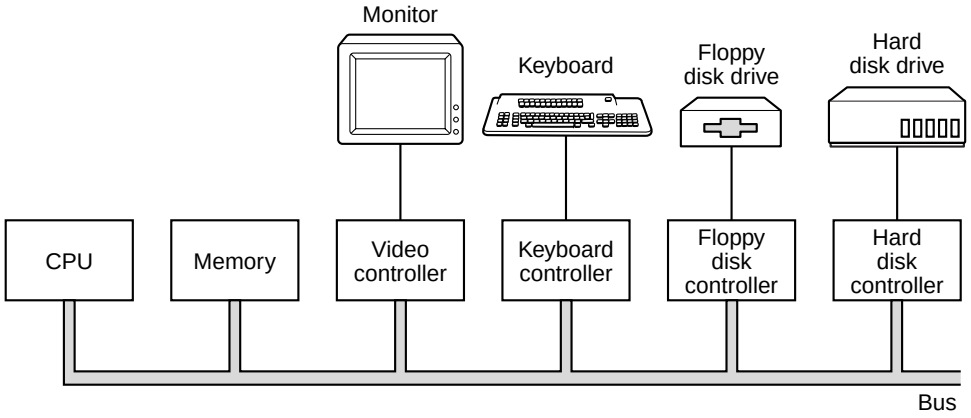


Figure 2-2. Logical structure of a simple personal computer.

The job of a controller is to control its I/O device and handle bus access for it. When a program wants data from the disk for example, it gives a command to the disk controller, which then issues seeks and other commands to the drive. When the proper track and sector have been located, the drive begins outputting the data as a serial bit stream to the controller. It is the job of the controller to break the bit stream up into units, and write each unit into memory, as it is assembled. A unit is typically one or more words. A controller that reads or writes data to or from memory without CPU intervention is said to be performing **Direct Memory Access**, better known by its acronym **DMA**. When the transfer is completed, the controller normally causes an **interrupt**, forcing the CPU to suspend running its current program and start running a special procedure, called an **interrupt handler**, to check for errors, take any special action needed, and inform the operating system that the I/O is now finished. When the interrupt handler is finished, the CPU continues with the program that was suspended when the interrupt occurred.

The bus is not only used by the I/O controllers, but also by the CPU for fetching instructions and data. What happens if the CPU and an I/O controller want to use the bus at the same time? The answer is that a chip called a **bus arbiter** decides who goes next. In general, I/O devices are given preference over the CPU, because disks and other moving devices cannot be stopped, and forcing them to wait would result in lost data. When no I/O is in progress, the CPU can have all the bus cycles for itself to reference memory. However, when some I/O device is also running, that device will request and be granted the bus when it needs it. This process is called **cycle stealing** and it slows down the computer.

This design worked fine for the first personal computers, since all the components were roughly in balance. However, as the CPUs, memories, and I/O devices got faster, a problem arose: the bus could no longer handle the load presented. On a closed system, such as an engineering workstation, the solution

was to design a new and faster bus for the next model. Because nobody ever moved I/O devices from an old model to a new one, this approached worked fine.

However, in the PC world, people often upgraded their CPU but wanted to move their printer, scanner, and modem to the new system. Also, a huge industry had grown up around providing a vast range of I/O devices for the IBM PC bus, and this industry had exceedingly little interest in throwing out its entire investment and starting over. IBM learned this the hard way when it brought out the successor to the IBM PC, the PS/2 range. The PS/2 had a new, and faster bus, but most clone makers continued to use the old PC bus, now called the **ISA** (**Industry Standard Architecture**) bus. Most disk and I/O device makers also continued to make controllers for it, so IBM found itself in the peculiar situation of being the only PC maker that was no longer IBM compatible. Eventually, it was forced back to supporting the ISA bus. As an aside, please note that ISA stands for Instruction Set Architecture in the context of machine levels whereas it stands for Industry Standard Architecture in the context of buses.

Nevertheless, despite the market pressure not to change anything, the old bus really was too slow, so something had to be done. This situation led to other companies developing machines with multiple buses, one of which was the old ISA bus, or its backward-compatible successor, the **EISA** (**Extended ISA**) bus. The most popular of these now is the **PCI** (**Peripheral Component Interconnect**) bus. It was designed by Intel, but Intel decided to put all the patents in the public domain, to encourage the entire industry (including its competitors) to adopt it.

The PCI bus can be used in many configurations, but a typical one is illustrated in Fig. 2-3. Here the CPU talks to a memory controller over a dedicated high-speed connection. The controller talks to the memory and to the PCI bus directly, so CPU-memory traffic does not go over the PCI bus. However, high-bandwidth (i.e., high data rate) peripherals, such as SCSI disks, can connect to the PCI bus directly. In addition, the PCI bus has a bridge to the ISA bus, so that ISA controllers and their devices can still be used. A machine of this design would typically contain three or four empty PCI slots and another three or four empty ISA slots, to allow customers to plug in both old ISA I/O cards (usually for slow devices) and new PCI I/O cards (usually for fast devices).

Many kinds of I/O devices are available today. A few of the more common ones are discussed below.

2.4.2 Terminals

Computer terminals consist of two parts: a keyboard and a monitor. In the mainframe world, these parts are often integrated into a single device and attached to the main computer by a serial line or over a telephone line. In the airline reservation, banking, and other mainframe-oriented industries, these devices are still in widespread use. In the personal computer world, the keyboard and monitor are independent devices. Either way, the technology of the two parts is the same.

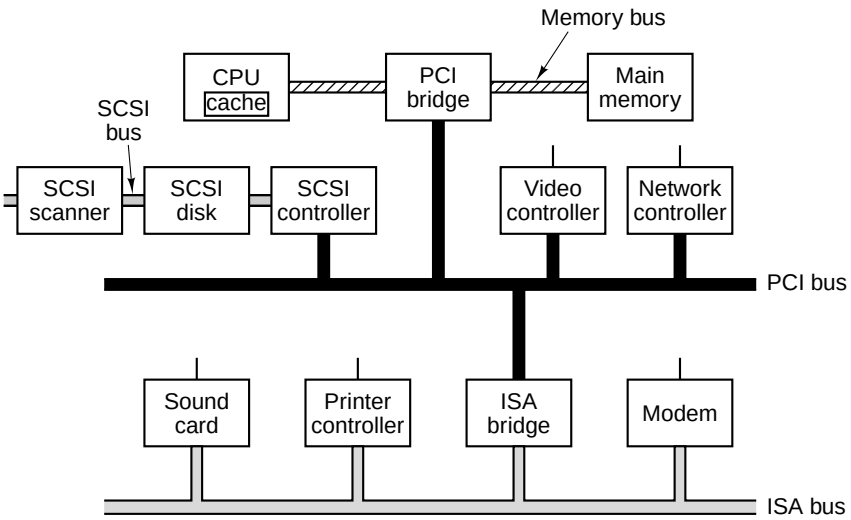


Figure 2-3. A typical modern PC with a PCI bus and an ISA bus. The modem and sound card are ISA devices; the SCSI controller is a PCI device.

Keyboards

Keyboards come in several varieties. The original IBM PC came with a keyboard that had a snap-action switch under each key that gave tactile feedback and made a click with the key was depressed far enough. Nowadays, the cheaper keyboards have keys that just make mechanical contact when depressed. Better ones have a sheet of elastometric material (a kind of rubber) between the keys and the underlying printed circuit board. Under each key is a small dome that buckles when depressed far enough. A small spot of conductive material inside the dome closes the circuit. Some keyboards have a magnet under each key that passes through a coil when struck, thus inducing a current that can be detected. Various other methods, both mechanical and electromagnetic, are also in use.

On personal computers, when a key is depressed, an interrupt is generated and the keyboard interrupt handler (a piece of software that is part of the operating system) is started. The interrupt handler reads a hardware register inside the keyboard controller to get the number of the key (1 through 102) that was just depressed. When a key is released, a second interrupt is caused. Thus if a user depresses the SHIFT key, then depresses and releases the M key, then releases the SHIFT key, the operating system can see that the user wants an upper case "M" rather than a lower case "m." Handling of multikey sequences involving SHIFT, CTRL, and ALT is done entirely in software (including the infamous CTRL-ALT-DEL key sequence that is used to reboot all IBM PCs and clones).

CRT Monitors

A monitor is a box containing a **CRT** (**Cathode Ray Tube**) and its power supplies. The CRT contains a gun that can shoot an electron beam against a phosphorescent screen near the front of the tube, as shown in Fig. 2-4(a). (Color monitors have three electron guns, one each for red, green and blue.) During the horizontal scan, the beam sweeps across the screen in about 50 μsec, tracing out an almost horizontal line on the screen. Then it executes a horizontal retrace to get back to the left-hand edge in order to begin the next sweep. A device like this that produces an image line by line is called a **raster scan** device.

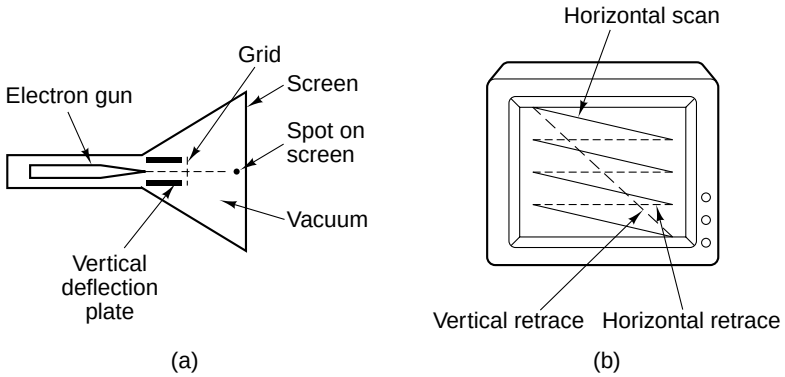


Figure 2-4. (a) Cross section of a CRT. (b) CRT scanning pattern.

Horizontal sweeping is controlled by a linearly increasing voltage applied to the horizontal deflection plates placed to the left and right of the electron gun. Vertical motion is controlled by a much more slowly linearly increasing voltage applied to the vertical deflection plates placed above and below the gun. After somewhere between 400 and 1000 sweeps, the voltages on the vertical and horizontal deflection plates are rapidly reversed together to put the beam back in the upper left-hand corner. A full-screen image is normally repainted between 30 and 60 times a second. The beam motions are shown in Fig. 2-4(b). Although we have described CRTs as using electric fields for sweeping the beam across the screen, many models use magnetic fields instead of electric ones, especially in high-end monitors.

To produce a pattern of dots on the screen, a grid is present inside the CRT. When a positive voltage is applied to the grid, the electrons are accelerated, causing the beam to hit the screen and make it glow briefly. When a negative voltage is used, the electrons are repelled, so they do not pass through the grid and the screen does not glow. Thus the voltage applied to the grid causes the corresponding bit pattern to appear on the screen. This mechanism allows a binary electrical signal to be converted into a visual display consisting of bright and dark spots.

Flat Panel Displays

CRTs are far too bulky and heavy to be used in notebook computers, so a completely different technology is needed for their screens. The most common one is **LCD** (**Liquid Crystal Display**) technology. It is highly complex, has many variations, and is changing rapidly, so this description will, of necessity, be brief and greatly simplified.

Liquid crystals are viscous organic molecules that flow like a liquid but also have spatial structure, like a crystal. They were discovered by an Austrian botanist (Rheinitzer) in 1888, and first applied to displays (e.g., calculators, watches) in the 1960s. When all the molecules are lined up in the same direction, the optical properties of the crystal depend on the direction and polarization of the incoming light. Using an applied electric field, the molecular alignment, hence the optical properties, can be changed. In particular, by shining a light through a liquid crystal, the intensity of the light exiting from it can be controlled electrically. This property can be exploited to construct flat panel displays.

An LCD display screen consists of two parallel glass plates between which is a sealed volume containing a liquid crystal. Transparent electrodes are attached to both plates. A light behind the rear plate (either natural or artificial) illuminates the screen from behind. The transparent electrodes attached to each plate used to create electric fields in the liquid crystal. Different parts of the screen get different voltages, to control the image displayed. Glued to the front and rear of the screen are polaroids because the display technology requires the use of polarized light. The general setup is shown in Fig. 2-5(a).

Although many kinds of LCD displays are in use, we will now consider one particular kind of display, the **TN** (**Twisted Nematic**) display as an example. In this display, the rear plate contains tiny horizontal grooves and the front plate contains tiny vertical grooves, as illustrated in Fig. 2-5(b). In the absence of an electric field, the LCD molecules tend to align with the grooves. Since the front and rear alignments differ by 90 degrees, the molecules (and thus the crystal structure) twist from rear to front.

At the rear of the display is a horizontal polaroid. It only allows in horizontally polarized light. At the front of the display is a vertical polaroid. It only allows vertically polarized light to pass through. If there were no liquid present between the plates, horizontally polarized light let in by the rear polaroid would be blocked by the front polaroid, making the screen uniformly black.

However the twisted crystal structure of the LCD molecules guides the light as it passes and rotates its polarization, making it come out horizontally. Thus in the absence of an electric field, the LCD screen is uniformly bright. By applying a voltage to selected parts of the plate, the twisted structure can be destroyed, blocking the light in those parts.

Two schemes are commonly used for applying the voltage. In a (low-cost) **passive matrix display**, both electrodes contain parallel wires. In a 640×480

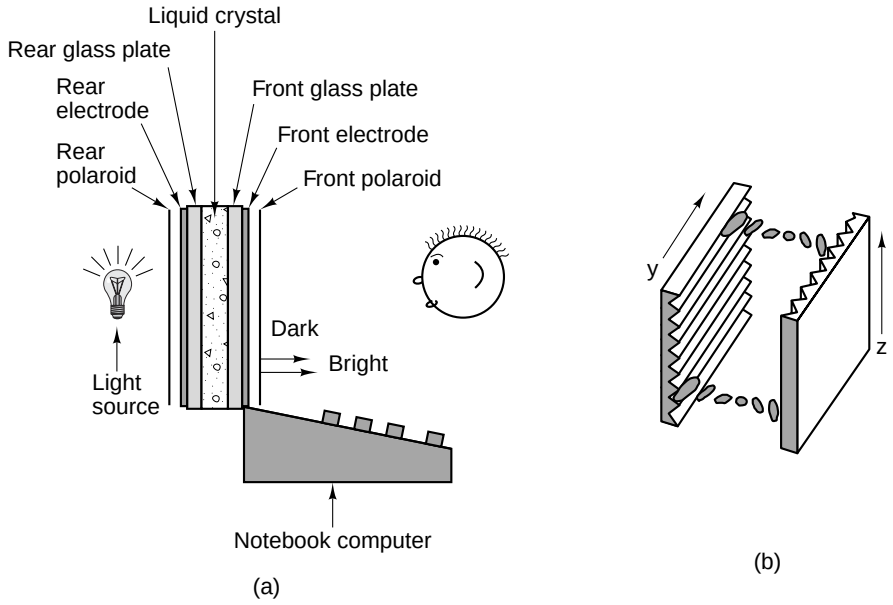


Figure 2-5. (a) The construction of an LCD screen. (b) The grooves on the rear and front plates are perpendicular to one another.

display, for example, the rear electrode might have 640 vertical wires and the front one might have 480 horizontal ones. By putting a voltage on one of the vertical wires and then pulsing one of the horizontal ones, the voltage at one selected pixel position can be changed, making it go dark briefly. By repeating this pulse with the next pixel and then the next one, a dark scan line can be painted, analogous to how a CRT works. Normally, the entire screen is painted 60 times a second to fool the eye into thinking there is a constant image there, again, the same way as a CRT.

The other scheme in widespread use is the **active matrix display**. It is considerably more expensive but it gives a better image so it is winning ground. Instead of just having two sets of perpendicular wires, it has a tiny switching element at each pixel position on one of the electrodes. By turning these on and off, an arbitrary voltage pattern can be created across the screen, allowing for an arbitrary bit pattern.

So far we have described how a monochrome display works. Suffice it to say that color displays uses the same general principles as monochrome displays, but that the details are a great deal more complicated. Optical filters are used to separate the white light into red, green, and blue components at each pixel position so these can be displayed independently. Every color can be built up from a linear superposition of these three primary colors.

Character-Map Terminals

Three kinds of terminals are in common use: character-map terminals, bit-map terminals, and RS-232-C terminals. They all can use any keyboard type, but they differ in the way the computer communicates with them and how the output is handled. We will now briefly describe each kind.

On a personal computer, there are two ways to organize the output to the screen: a character map and a bit map. Figure 2-6 shows how a character map is used to display output on the monitor. (The keyboard is treated as a completely separate device.) On the serial communication board is a chunk of memory, called the **video memory**, as well as some electronics for accessing the bus and generating video signals.

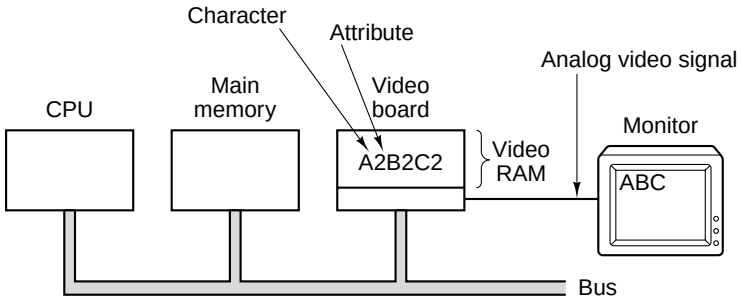


Figure 2-6. Terminal output on a personal computer.

To display characters, the CPU copies them to the video memory in alternate bytes. Associated with each character is an **attribute byte** that describes how that character is to be displayed. Attributes can include its color, intensity, whether it is blinking, and so on. Thus a screen image of 25×80 characters requires 4000 bytes of video memory, 2000 for the characters and 2000 for the attributes. Most boards have more memory to hold multiple screen images.

The job of the video board is to repeatedly fetch characters from the video RAM and generate the necessary signal to drive the monitor. An entire line of characters is fetched at once so the individual scan lines can be computed. This signal is a high-frequency analog signal that controls the scanning of the electron beam that paints the characters on the screen. Because the board outputs a video signal, the monitor must be within a few meters of the computer to prevent distortion.

Bit-map Terminals

A variation on this idea is to have the screen not be regarded as a 25×80 array of characters, but as an array of picture elements, called **pixels**. Each pixel is either on or off. It represents one bit of information. On personal computers the

screen may contain as few as 640×480 pixels, but more commonly 800×600 or more. On engineering workstations, the screen is typically 1280×960 pixels or more. Terminals using a bit map rather than a character map are called **bit-map terminals**. All modern video boards can operate either as character-map terminals or bit-map terminals, under software control.

The same general idea is used as in Fig. 2-6, except that the video RAM is just seen as a big bit array. The software can set any pattern it wants there, and that is displayed instantly. To draw characters, the software might decide to allocate, for example, a 9 by 14 rectangle for each character and fill in the necessary bits to make the character appear. This approach allows the software to create multiple fonts and intermix them at will. All the hardware does is display the bit array. For color displays, each pixel is 8, 16, or 24 bits.

Bit-map terminals are commonly used to support displays containing several **windows**. A window is an area of the screen used by one program. With multiple windows, it is possible to have several programs running at the same time, each one displaying its results independent of the other ones.

Although bit-map terminals are highly flexible, they have two major disadvantages. First, they require a considerable amount of video RAM. The most common sizes these days are 640×480 (VGA), 800×600 (SVGA), 1024×768 (XVGA), and 1280×960). Notice that all of these have an aspect ratio (width:height) of 4:3, to conform to the current ratio for television sets. To get true color, 8 bits are needed for each of the three primary colors, or 3 bytes/pixel. Thus a 1024×768 display, requires 2.3 MB of video RAM.

As a result of this large requirement, some computers compromise by using an 8-bit number to indicate the color desired. This number is then used as an index into a hardware table, called the **color palette** that contains 256 entries, each holding a 24-bit RGB value. Such a design, called **indexed color**, reduces the memory video RAM memory requirements by 2/3, but allows only 256 colors on the screen at once. Usually, each window on the screen has its own mapping, but with only one hardware color palette, often when multiple windows are present on the screen, only the current one has its colors rendered correctly.

The second disadvantage of a bit-map display is performance. Once application programmers realize that they can control every pixel in both space and time, they want to do it. Although data can be copied from the video RAM to the monitor without going over the main system bus, getting data into the video RAM does use the system bus. To display full-screen, full-color multimedia on a 1024×768 display requires copying 2.3 MB of data to the video RAM for every frame. For full-motion video, a rate of at least 25 frame/sec is needed, for a total data rate of 57.6 MB/sec. This load is far more than what the (E)ISA bus can handle, so high-performance video cards on IBM PCs need to be PCI cards, and even then, major compromises are required.

A related performance problem is how to scroll the screen. One way is copying all the bits in software, but doing this puts a gigantic load on the CPU. Not

surprisingly, many video cards are equipped with special hardware for moving parts of the screen by changing base registers rather than copying.

RS-232-C Terminals

Dozens of companies make computers and many others make terminals (especially for mainframes). To allow (almost) any terminal to be used with (almost) any computer, a standard computer-terminal interface, called **RS-232-C**, has been developed by the Electronics Industries Association (EIA). Any terminal that supports the RS-232-C interface can be connected to any computer that also supports this interface.

RS-232-C terminals have a standardized 25-pin connector on them. The RS-232-C standard defines the mechanical size and shape of the connector, the voltage levels, and the meaning of each of the signals on the pins.

When the computer and the terminal are far apart, it is frequently the case that the only practical way to connect them is over the telephone system. Unfortunately, the telephone system is not capable of transmitting the signals required by the RS-232-C standard, so a device called a **modem** (**mo**dulator-**dem**odulator) has to be inserted between the computer and the telephone and also between the terminal and the telephone to perform signal conversion. We will study modems shortly.

Figure 2-7 shows the placement of the computer, modems, and terminal when a telephone line is used. When the terminal is close enough to the computer that it can be wired-up directly, modems are not used, but the same RS-232-C connectors and cables are still used, although those pins related to modem control are not needed.

To communicate, the computer and terminal each contain a chip called a **UART** (**Universal Asynchronous Receiver Transmitter**), as well as logic to access the bus. To display a character, the computer fetches a character from its main memory and presents it to the UART, which then shifts it out onto the RS-232-C cable bit-for-bit. In effect, the UART is really a parallel-to-serial converter, since an entire character (1 byte) is given to it at once, and it outputs the bits one at a time at a specific rate. It also adds a start bit and a stop bit to each character to delimit the beginning and the end of the character (at 110 bps, 2 stop bits are used).

In the terminal, another UART receives the bits and rebuilds the entire character, which is then displayed on the screen. Input from the terminal's keyboard goes through a parallel-to-serial conversion in the terminal and is then reassembled by the UART in the computer.

The RS-232-C standard defines almost 25 signals, but in practice, only a few are used (and most of those can be omitted when the terminal is wired directly to the computer without modems). Pins 2 and 3 are for transmitting and receiving data, respectively. Each pin handles a one-way bit stream, in opposite directions.

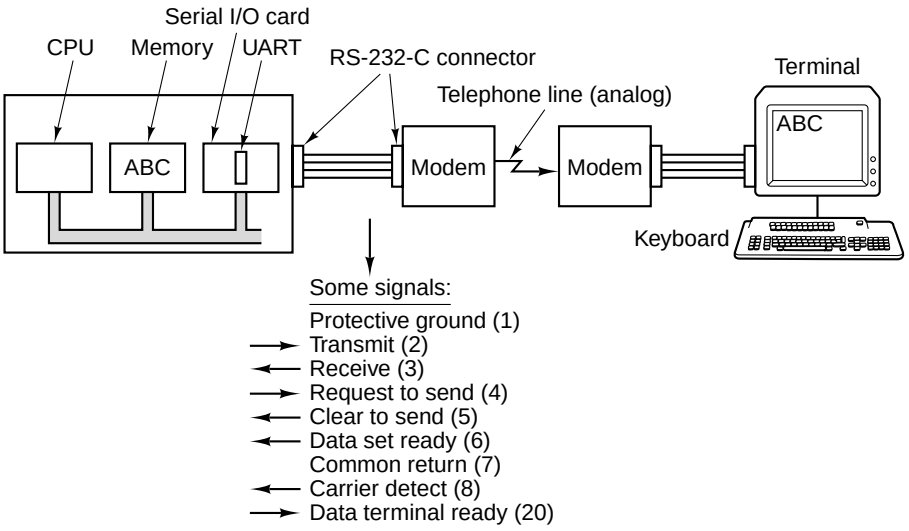


Figure 2-7. Connection of an RS-232-C terminal to a computer. The numbers in parentheses in the list of signals are the pin numbers.

When the terminal or computer is powered up, it asserts (i.e., sets to 1) the Data Terminal Ready signal to inform the modem that it is on. Similarly, the modem asserts Data Set Ready to signal its presence. When the terminal or computer wants to send data, it asserts Request to Send to ask permission. If the modem is willing to grant that permission, it asserts Clear to Send as a response. Other pins are used for various status, testing, and timing functions.

2.4.3 Mice

As time goes on, computers are being used by people with less expertise in how computers work. Computers of the ENIAC generation were used only by the people who built them. In the 1950s, computers were only used by highly-skilled professional programmers. Now, computers are widely used by people who need to get some job done and do not know (or even want to know) much about how computers work or how they are programmed.

In the old days, most computers had command line interfaces, to which users typed commands. Since people who are not computer specialists often perceived command line interfaces as user-unfriendly, if not downright hostile, many computer vendors developed point-and-click interfaces, such as the Macintosh and Windows. Using this model requires having a way to point at the screen. The most common way of allowing users to point at the screen is with a mouse.

A **mouse** is a small plastic box that sits on the table next to the keyboard. When it is moved around on the table, a little pointer on the screen moves too,

allowing users to point at screen items. The mouse has one, two, or three buttons on top, to allow users to select items from menus. Much blood has been spilled as a result of arguments about how many buttons a mouse ought to have. Naive users prefer one (it is hard to push the wrong button if there is only one), but sophisticated ones like the power of multiple buttons to do fancy things.

Three kinds of mice have been produced: mechanical mice, optical mice, and optomechanical mice. The first mice had two rubber wheels protruding through the bottom, with their axles perpendicular to one another. When the mouse was moved parallel to its main axis, one wheel turned. When it is moved perpendicular to its main axis, the other one turned. Each wheel drove a variable resistor (potentiometer). By measuring changes in the resistance, it was possible to see how much each wheel had rotated and thus calculate how far the mouse had moved in each direction. In recent years, this design has largely been replaced by one in which a ball that protrudes slightly from the bottom is used instead of wheels. It is shown in Fig. 2-8.

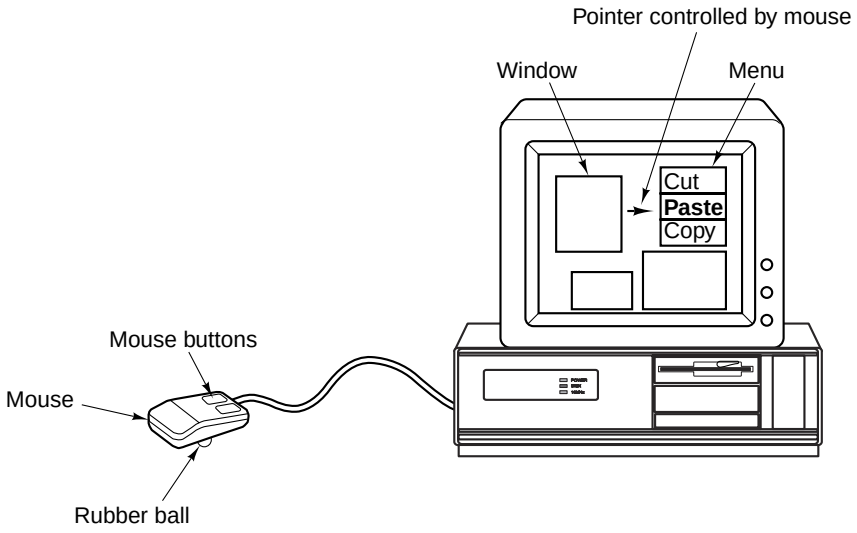


Figure 2-8. A mouse being used to point to menu items.

The second kind of mouse is the optical mouse. This kind has no wheels or ball. Instead, it has an **LED** (**Light Emitting Diode**) and a photodetector on the bottom. The optical mouse is used on top of a special plastic pad containing a rectangular grid of closely spaced lines. As the mouse moves over the grid, the photodetector senses line crossings by seeing the changes in the amount of light being reflected back from the LED. Electronics inside the mouse count the number of grid lines crossed in each direction.

The third kind of mouse is optomechanical. Like the newer mechanical mouse, it has a rolling ball that turns two shafts aligned at 90 degrees to each

other. The shafts are connected to encoders that have slits through which light can pass. As the mouse moves, the shafts rotate, and light pulses strike the detectors whenever a slit comes between an LED and its detector. The number of pulses detected is proportional to the amount of motion.

Although mice can be set up in various ways, a common arrangement is to have the mouse send a sequence of 3 bytes to the computer every time the mouse moves a certain minimum distance (e.g., 0.01 inch), sometimes called a **mickey**. Usually, these characters come in on a serial line, one bit at time. The first byte contains a signed integer telling how many units the mouse has moved in the *x*-direction in the last 100 msec. The second byte gives the same information for *y* motion. The third byte contains the current state of the mouse buttons. Sometimes 2 bytes are used for each coordinate.

Low-level software in the computer accepts this information as it comes in and converts the relative movements sent by the mouse to an absolute position. It then displays an arrow on the screen at the position corresponding to where the mouse is. When the arrow points at the proper item, the user clicks a mouse button, and the computer can then figure out which item has been selected from its knowledge of where the arrow is on the screen.

2.4.4 Printers

Having prepared a document or fetched a page from the World Wide Web, users often want to print it, so all computers can be equipped with a printer. In this section we will describe some of the more common kinds of monochrome (i.e., black and white) and color printers.

Monochrome Printers

The cheapest kind of printer is the **matrix printer**, in which a print head containing between 7 and 24 electromagnetically activatable needles is scanned across each print line. Low-end printers have seven needles, for printing, say, 80 characters in a 5×7 matrix across the line. In effect, the print line then consists of 7 horizontal lines, each consisting of $5 \times 80 = 400$ dots. Each dot can be printed or not printed, depending on the characters to be printed. Figure 2-9(a) illustrates the letter "A" printed on a 5×7 matrix.

The print quality can be increased by two techniques: using more needles and having the circles overlap. Figure 2-9(b) shows an "A" printed using 24 needles that produce overlapping dots. Usually, multiple passes over each scan line are required to produce overlapping dots, so increased quality goes hand in hand with slower printing rates. Most matrix printers can operate in several modes, offering different trade-offs between print quality and speed.

Matrix printers are cheap (especially in terms of consumables) and highly reliable, but slow, noisy, and poor at graphics. They have three main uses in current

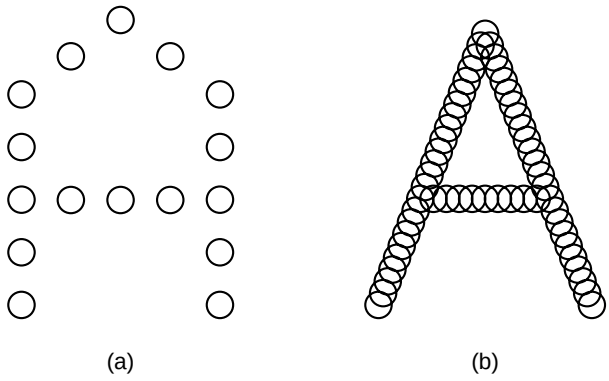


Figure 2-9. (a) The letter "A" on a 5×7 matrix. (b) The letter "A" printed with 24 overlapping needles.

systems. First, they are popular for printing on large (> 30 cm) preprinted forms. Second, they are good at printing on small pieces of paper, such as cash register receipts, ATM machine or credit card transaction slips, or airline boarding passes. Third, for printing on multipart continuous forms with carbon paper embedded between the copies, they are usually the cheapest technology.

For low-cost home printing, **inkjet printers** are a favorite. The movable print head, which holds an ink cartridge, is swept horizontally across the paper while ink is sprayed from its tiny nozzles. Inside each nozzle, an ink droplet is electrically heated to the boiling point until it explodes. The only place the ink can go is out the front of the nozzle onto the paper. The nozzle is then cooled and the resulting vacuum sucks in another ink droplet. The speed of the printer is limited by how fast the boil/cool cycle can be repeated. Inkjet printers typically have resolutions of 300 **dpi** (**dots per inch**) to 720 dpi, although 1440 dpi inkjet printers are also available. They are cheap, quiet, and have good quality, although they are also slow, use expensive ink cartridges, and produce ink-soaked output.

Probably the most exciting development in printing since Johann Gutenberg invented movable type in the fifteenth century is the **laser printer**. This device combines a high quality image, excellent flexibility, good speed, and moderate cost into a single peripheral. Laser printers use almost the same technology as photocopy machines. In fact, many companies make devices that combine copying and printing (and sometimes fax as well).

The basic technology is illustrated in Fig. 2-10. The heart of the printer is a rotating precision drum (or in some high-end systems, a belt). At the start of each page cycle, it is charged up to about 1000 volts and coated with a photosensitive material. Then light from a laser is scanned along the length of the drum much like the electron beam in a CRT, only instead of achieving the horizontal deflection using a voltage, a rotating octagonal mirror is used to scan the length of the

drum. The light beam is modulated to produce a pattern of light and dark spots. The spots where the beam hits lose their electrical charge.

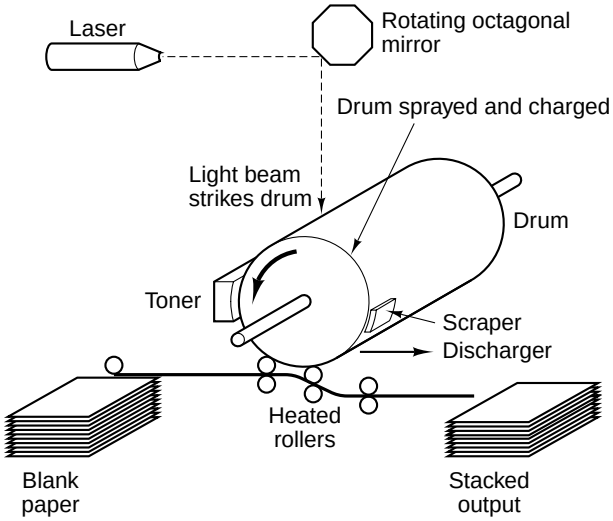


Figure 2-10. Operation of a laser printer.

After a line of dots has been painted, the drum rotates a fraction of a degree to allow the next line to be painted. Eventually, the first line of dots reaches the toner, a reservoir of an electrostatically sensitive black powder. The toner is attracted to those dots that are still charged, thus forming a visual image of that line. A little later in the transport path, the toner-coated drum is pressed against the paper, transferring the black powder to the paper. The paper is then passed through heated rollers to fuse the toner to the paper permanently, fixing the image. Later in its rotation, the drum is discharged and scraped clean of any residual toner, preparing it for being charged and coated again for the next page.

That this process is an exceedingly complex combination of physics, chemistry, mechanical engineering, and optical engineering hardly needs to be said. Nevertheless, complete assemblies, called **print engines**, are available from several vendors. Laser printer manufacturers combine the print engines with their own electronics and software to make a complete printer. The electronics consists of a fast CPU along with megabytes of memory to hold a full-page bit map and numerous fonts, some of them built in and some of them downloadable. Most printers accept commands that describe the pages to be printed (as opposed to simply accepting bit maps prepared by the main CPU). These commands are given in languages such as HP's PCL and Adobe's PostScript.

Laser printers at 600 dpi and up can do a reasonable job of printing black and white photographs but the technology is trickier than it might at first appear. Consider a photograph scanned in at 600 dpi that is to be printed on a 600 dpi printer.

The scanned image contains 600×600 pixels/inch, each one consisting of a gray value from 0 (white) to 255 (black). The printer can also print 600 dpi, but each printed pixel is either black (toner present) or white (no toner present). Gray values cannot be printed.

The usual solution to printing images with gray values is to use **halftoning**, the same as commercially printed posters. The image is broken up into halftone cells, each typically 6×6 pixels. Each cell can contain between 0 and 36 black pixels. The eye perceives a cell with many pixels as darker than one with fewer pixels. Gray values in the range 0 to 255 are represented by dividing this range into 37 zones. Values from 0 to 6 are in zone 0, values from 7 to 13 are in zone 1, and so on (zone 36 is slightly smaller than the others because 37 does not divide 256 exactly). Whenever a gray value in zone 0 is encountered, its halftone cell on the paper is left blank, as illustrated in Fig. 2-11(a). A zone 1 value is printed as 1 black pixel. A zone 2 value is printed as 2 black pixels, as shown in Fig. 2-11(b). Other zone values are shown in Fig. 2-11(c)-(f). Of course, taking a photograph scanned at 600 dpi and halftoning this way reduces the effective resolution to 100 cells/inch, called the **halftone screen frequency**, conventionally measured in **lpi** (**lines per inch**).

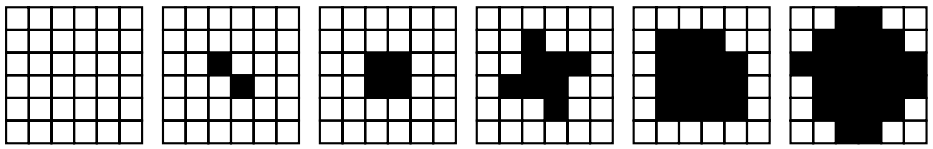


Figure 2-11. Halftone dots for various gray scale ranges. (a) 0–6. (b) 14–20. (c) 28–34. (d) 56–62. (e) 105–111. (f) 161–167.

Color Printers

Color images can be viewed in one of two ways: transmitted light and reflected light. Transmitted light images, such as those produced on CRT monitors, are built up from the linear superposition of the three additive primary colors, red, green, and blue. Reflected light images, such as color photographs and pictures in glossy magazines, absorb certain wavelengths of light and reflect the rest. These are built up from a linear superposition of the three subtractive primary colors, cyan (all red absorbed), yellow (all blue absorbed), and magenta (all green absorbed). In theory, every color can be produced by mixing cyan, yellow, and magenta ink. In practice it is difficult to get the inks pure enough to absorb all light and produce a true black. For this reason, nearly all color printing systems use four inks: cyan, yellow, magenta, and black. These systems are called **CYMK printers** (K is for blacK, to avoid confusion with Blue). Monitors, in contrast, use transmitted light and the RGB system for producing colors.

The complete set of colors that a display or printer can produce is called its **gamut**. No device has a gamut that matches the real world, since at best each color comes in 256 intensities, giving only 16,777,216 discrete colors. Imperfections in the technology reduce the total more, and the remaining ones are not always uniformly spaced over the color spectrum. Furthermore, color perception has a lot to do with how the rods and cones in the human retina work, and not just the physics of light.

As a consequence of the above observations, converting a color image that looks fine on the screen to an identical printed one is far from trivial. Among the problems are

1. Color monitors use transmitted light; color printers use reflected light.
2. CRTs produce 256 intensities per color; color printers must halftone.
3. Monitors have a dark background; paper has a light background.
4. The RGB and CMYK gamuts are different.

Getting printed color images to match real life (or even to match screen images) requires device calibration, sophisticated software, and considerable expertise on the part of the user.

Five technologies are in common use for color printing, all of them based on the CMYK system. At the low end are color ink jet printers. They work the same way as monochrome ink jet printers, but with four cartridges (for C, M, Y, and K) instead of one. They give good results for color graphics and passable results for photographs at modest cost (the printers are cheap but the ink cartridges are not).

For best results, special ink and paper should be used. Two kinds of ink exist. **Dye-based inks** consist of colored dyes dissolved in a fluid carrier. They give bright colors and flow easily. Their main disadvantage is that they fade when exposed to ultraviolet light, such as that contained in sunlight. **Pigment-based ink** contains solid particles of pigment suspended in a fluid carrier that evaporates from the paper, leaving the pigment behind. They do not fade in time but are not as bright as dye-based inks and the pigment particles have a tendency to clog the nozzles, requiring periodic cleaning. Coated or glossy paper is required for printing photographs. These kinds of paper have been specially designed to hold the ink droplets and not let them spread out.

A step up from ink jet printers leads to the **solid ink printers**. These accept four solid blocks of a special waxy ink which are then melted into hot ink reservoirs. Startup times of these printers can be as much as 10 minutes, while the ink blocks are melting. The hot ink is sprayed onto the paper, where it solidifies and is fused with the paper by forcing it between two hard rollers

The third kind of color printer is the color laser printer. It works like its monochrome cousin, except that separate C, Y, M, and K images are laid down and transferred to a roller using four different toners. Since the full bit map is

generally produced in advance, a 1200×1200 dpi image for a page containing 80 square inches needs 115 million pixels. With 4 bits/pixels, the printer needs 55 MB just for the bit map, exclusive of memory for the internal processors, fonts, etc. This requirement makes color laser printers expensive, but printing is fast, the quality is high, and the images are stable over time.

The fourth kind of color printer is the **wax printer**. It has a wide ribbon of four-color wax that is segmented into page-size bands. Thousands of heating elements melt the wax as the paper moves under it. The wax is fused to the paper in the form of pixels using the CMYK system. Wax printers used to be the main color printing technology, but they are being replaced by the other kinds, which have cheaper consumables.

The fifth kind of color printer is the **dye sublimation printer**. Although it has Freudian undertones, sublimation is the scientific name for a solid changing into a gas without passing through the liquid state. Dry ice (frozen carbon dioxide) is a well-known material that subliminates. In a dye sublimation printer, a carrier containing the CMYK dyes passes over a thermal print head containing thousands of programmable heating elements. The dyes are vaporized instantly and absorbed by a special paper close by. Each heating element can produce 256 different temperatures. The higher the temperature, the more dye that is deposited and the more intense the color. Unlike all the other color printers, nearly continuous colors are possible for each pixel, so no halftoning is needed. Small snapshot printers often use the dye sublimation process to produce highly realistic photographic images on special (and expensive) paper.

2.4.5 Modems

With the growth of computer usage in the past years, it is common for one computer to need to communicate with another one. For example, many people have personal computers at home that they use for communicating with their computer at work, with an Internet Service Provider, or with a home banking system. All these applications use the telephone to provide the underlying communication.

However, a raw telephone line is not suitable for transmitting computer signals, which generally represent a 0 as 0 volts and a 1 as 3 to 5 5 volts as shown in Fig. 2-12(a). Two-level signals suffer considerable distortion when transmitted over a voice-grade telephone line, thereby leading to transmission errors. A pure sine wave signal at a frequency of 1000 to 2000 Hz, called a **carrier**, can be transmitted with relatively little distortion, however, and this fact is exploited as the basis of most telecommunication systems.

Because the pulsations of a sine wave are completely predictable, a pure sine wave transmits no information at all. However, by varying the amplitude, frequency, or phase, a sequence of 1s and 0s can be transmitted, as shown in Fig. 2-12. This process is called **modulation**. In **amplitude modulation** [see Fig. 2-12(b)], two different voltage levels are used, for 0 and 1, respectively. A person

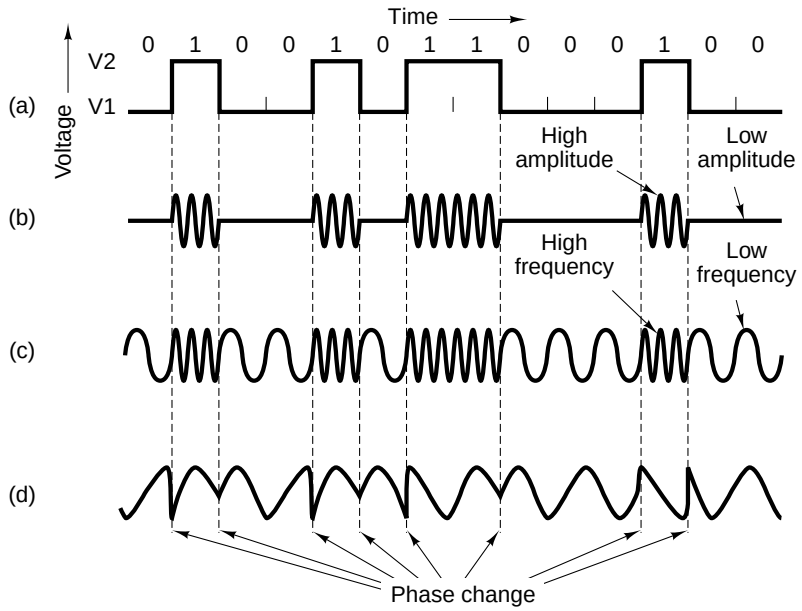


Figure 2-12. Transmission of the binary number 01001011000100 over a telephone line bit by bit. (a) Two-level signal. (b) Amplitude modulation. (c) Frequency modulation. (d) Phase modulation.

listening to digital data transmitted at a very low data rate would hear a loud noise for a 1 and no noise for a 0.

In **frequency modulation** [see Fig. 2-12(c)], the voltage level is constant but the carrier frequency is different for 1 and 0. A person listening to frequency modulated digital data would hear two tones, corresponding to 0 and 1. Frequency modulation is often referred to as **frequency shift keying**.

In simple **phase modulation** [see Fig. 2-12(d)], the amplitude and frequency do not change, but the phase of the carrier is reversed 180 degrees when the data switch from 0 to 1 or 1 to 0. In more sophisticated phase-modulated systems, at the start of each indivisible time interval, the phase of the carrier is abruptly shifted by 45, 135, 225, or 315 degrees, to allow 2 bits per time interval, called **dibit** phase encoding. For example, a phase shift of 45 degrees could represent 00, a phase shift of 135 degrees could represent 01, and so on. Other schemes, for transmitting 3 or more bits per time interval also exist. The number of time intervals (i.e., the number of potential signal changes per second) is **baud** rate. With 2 or more bits per interval, the bit rate will exceed the baud rate. Many people confuse these two terms.

If the data to be transmitted consist of a series of 8-bit characters, it would be desirable to have a connection capable of transmitting 8 bits simultaneously—that is, eight pairs of wires. Because voice-grade telephone lines provide only one

channel, the bits must be sent serially, one after another (or in groups of two if dibit encoding is being used). The device that accepts characters from a computer in the form of two-level signals, one bit at a time, and transmits the bits in groups of one or two, in amplitude-, frequency-, or phase-modulated form, is the modem. To mark the start and end of each character, an 8-bit character is normally sent preceded by a start bit and followed by a stop bit, making 10 bits in all.

The transmitting modem sends the individual bits within one character at regularly-spaced time intervals. For example, 9600 baud implies one signal change every 104 μsec. A second modem at the receiving end is used to convert a modulated carrier to a binary number. Because the bits arrive at the receiver at regularly-spaced intervals, once the receiving modem has determined the start of the character, its clock tells it when to sample the line to read the incoming bits.

Modern modems operate at data rates ranging from 28,800 bits/sec to 57,600 bits/sec, usually at much lower baud rates. They use a combination of techniques to send multiple bits per baud, modulating the amplitude, frequency, and phase. Nearly all of them are **full-duplex**, meaning they can transmit in both directions at the same time (using different frequencies). Modems or transmission lines that can only transmit in one direction at a time (like a single-track railroad that can handle north-bound trains or south-bound trains but not at the same time) are called **half-duplex**. Lines that can only transmit in one direction are **simplex**.

ISDN

In the early 1980s, the European PTTs (Post, Telephone, and Telegraph Administrations) developed a standard for digital telephony called **ISDN** (**Integrated Services Digital Network**). It was intended to allow senior citizens to have alarms in their houses connected to central monitoring offices, and a lot of other strange applications that never happened. They pushed the idea hard, but without much success. Then all of sudden, the World Wide Web happened and people were clamoring for high-bandwidth digital access to the Internet. Bingo. ISDN suddenly discovered its killer application (through no fault of its designers, however). It also became popular in the United States and elsewhere.

When a telco (industry jargon for telephone company) customer subscribes to ISDN, the telco replaces the old analog line with a digital one. (Actually, the line itself is not replaced, just the equipment on both ends.) The new line holds two independent digital channels at 64,000 bits/sec each, plus a signaling channel at 16,000 bits/sec. Equipment exists to combine all three channels into a single 144,000 bps digital channel. For businesses, a 30-channel ISDN line is available.

Not only is ISDN faster than an analog channel, but it also allows connections to be established in typically no more than 1 sec, no longer requires an analog modem, and is much more reliable (fewer errors) than an analog line. It also has a variety of extra features and options not always available with analog lines.

The structure of an ISDN connection is shown in Fig. 2-13. What the carrier provides is a digital bit pipe that just moves bits. What the bits mean is up to the sender and receiver. The interface between the customer's equipment and the carrier's equipment is the NT1 device, with the T interface on one side and the U interface on the other. In the U.S. customers must buy their own NT1 device. In many European countries, they must rent it from the carrier.

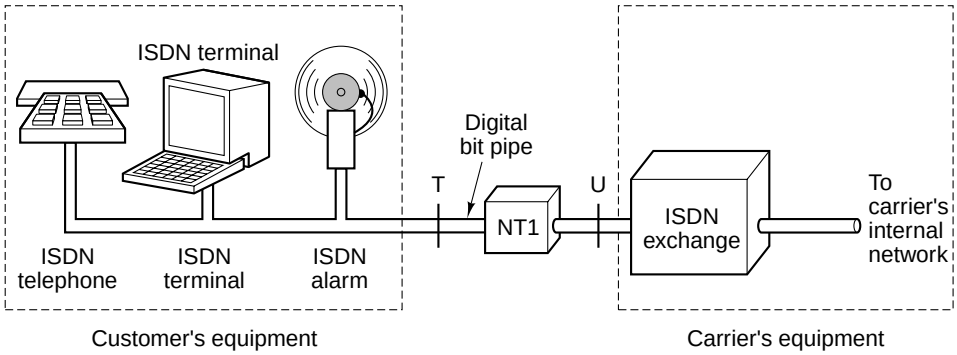


Figure 2-13. ISDN for home use.

2.4.6 Character Codes

Each computer has a set of characters that it uses. As a bare minimum, this set includes the 26 upper case letters, the 26 lower case letters, the digits 0 through 9, and a set of special symbols. such as space, period, minus sign, comma, and carriage return.

In order to transfer these characters into the computer, each one is assigned a number: for example, a = 1, b = 2, ..., z = 26, + = 27, – = 28. The mapping of characters onto integers is called a **character code**. It is essential that communicating computers use the same code or they will not be able to understand one another. For this reason, standards have been developed. Below we will examine two of the most important ones.

ASCII

One widely used code is called **ASCII** (**American Standard Code for Information Interchange**). Each ASCII character has 7 bits, allowing for 128 characters in all. Figure 2-14 shows the ASCII code. Codes 0 to 1F (hexadecimal) are control characters and do not print.

Many of the ASCII control characters are intended for data transmission. For example, a message might consist of an SOH (Start of Header) character, a header, an STX (Start of Text) character, the text itself, an ETX (End of Text)

Hex	Name	Meaning	Hex	Name	Meaning
0	NUL	Null	10	DLE	Data Link Escape
1	SOH	Start Of Heading	11	DC1	Device Control 1
2	STX	Start Of Text	12	DC2	Device Control 2
3	ETX	End Of Text	13	DC3	Device Control 3
4	EOT	End Of Transmission	14	DC4	Device Control 4
5	ENQ	Enquiry	15	NAK	Negative AcKnowledgement
6	ACK	ACKnowledgement	16	SYN	SYNchronous idle
7	BEL	BELl	17	ETB	End of Transmission Block
8	BS	BackSpace	18	CAN	CANcel
9	HT	Horizontal Tab	19	EM	End of Medium
A	LF	Line Feed	1A	SUB	SUBstitute
B	VT	Vertical Tab	1B	ESC	ESCape
C	FF	Form Feed	1C	FS	File Separator
D	CR	Carriage Return	1D	GS	Group Separator
E	SO	Shift Out	1E	RS	Record Separator
F	SI	Shift In	1F	US	Unit Separator

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char
20	(Space)	30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	’	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Figure 2-14. The ASCII character set.

character and then an EOT (End of Transmission) character. In practice, however, the messages sent over telephone lines and networks are formatted quite differently, so the ASCII transmission control characters are not used much any more.

The ASCII printing characters are straightforward. They include the upper and lower case letters, digits, punctuation marks and a few math symbols.

UNICODE

The computer industry grew up mostly in the U.S., which led to the ASCII character set. ASCII is fine for English but less fine for other languages. French needs accents (e.g., système); German needs diacritical marks (e.g., für), and so on. Some European languages have a few letters not found in ASCII, such as the German ß and the Danish ø. Some languages have entirely different alphabets (e.g., Russian and Arabic), and a few languages have no alphabet at all (e.g., Chinese). As computers spread to the four corners of the globe, and software vendors want to sell products in countries where most users do not speak English, a different character set is needed.

The first attempt at extending ASCII was IS 646, which added another 128 characters to ASCII, making it an 8-bit code called **Latin-1**. The additional characters were mostly Latin letters with accents and diacritical marks. The next attempt was IS 8859, which introduced the concept of a **code page**, a set of 256 characters for a particular language or group of languages. IS 8859-1 is Latin-1. IS 8859-2 handles the Latin-based Slavic languages (e.g., Czech, Polish, and Hungarian). IS 8859-3 contains the characters needed for Turkish, Maltese, Esperanto, and Galician, and so on. The trouble with the code page approach is that the software has to keep track of which page it is on, it is impossible to mix languages over pages, and the scheme does not cover Japanese and Chinese at all.

A group of computer companies decided to solve this problem by forming a consortium to create a new system, called **UNICODE**, and getting it proclaimed an International Standard (IS 10646). UNICODE is now supported by some programming languages (e.g., Java), some operating systems (e.g., Windows NT), and many applications. It is likely to become increasingly accepted as the computer industry goes global.

The basic idea behind UNICODE is to assign every character and symbol a unique, permanent 16-bit value, called a **code point**. No multibyte characters and no escape sequences are used. Having every symbol be 16 bits makes writing software much simpler.

With 16-bit symbols, UNICODE has 65,536 code points. Since the world's languages collectively use about 200,000 symbols, code points are a scarce resource that must be allocated with great care. About half the code points have already been allocated, and the UNICODE consortium is continually reviewing proposals to eat up the rest. To speed the acceptance of UNICODE, the consortium cleverly used Latin-1 as code points 0 to 255, making conversion between ASCII and UNICODE easy.

To avoid wasting code points, each diacritical mark has its own code point. It is up to software to combine diacritical marks with their neighbors to form new characters.

The code point space is divided up into blocks, each one a multiple of 16 code points. Each major alphabet in UNICODE has a sequence of consecutive zones.

Some examples (and the number of code points allocated) are Latin (336), Greek (144), Cyrillic (256), Armenian (96), Hebrew (112), Devanagari (128), Gurmukhi (128), Oriya (128), Telugu (128), and Kannada (128). Note that each of these languages has been allocated more code points than it has letters. This choice was made in part because many languages have multiple forms for each letter. For example, each letter in English has two forms—lower case and UPPER CASE. Some languages have three or more forms, possibly depending on whether the letter is at the start, middle, or end of a word.

In addition to these alphabets, code points have been allocated for diacritical marks (112), punctuation marks (112), subscripts and superscripts (48), currency symbols (48), mathematical symbols (256), geometric shapes (96), and dingbats (192).

After these come the symbols needed for Chinese, Japanese, and Korean. First are 1024 phonetic symbols (e.g., katakana and bopomofo) and then the unified Han ideographs (20,992) used in Chinese and Japanese, and the Korean Hangul syllables (11,156).

To allow users to invent special characters for special purposes, 6400 code points have been allocated for local use.

While UNICODE solves many problems associated with internationalization, it does not (attempt to) solve all the world's problems. For example, while the Latin alphabet is in order, the Han ideographs are not in dictionary order. As a consequence, an English program can examine ''cat'' and ''dog'' and sort them alphabetically by simply comparing the UNICODE value of their first character. A Japanese program needs external tables to figure out which of two symbols comes before the other in the dictionary.

Another issue is that new words are popping up all the time. Fifty years ago nobody talked about applets, cyberspace, gigabytes, lasers, modems, smileys, or videotapes. Adding new words in English does not require new code points. Adding them in Japanese does. In addition to new technical words, there is a demand for adding at least 20,000 new (mostly Chinese) personal and place names. Blind people think Braille should be in there, and special interest groups of all kinds want what they perceive as their rightful code points. The UNICODE consortium reviews and decides on all new proposals.

UNICODE uses the same code point for characters that look almost identical but have different meanings or are written slightly differently in Japanese and Chinese (as though English word processors always spelled ''blue'' as ''blew'' because they sound the same). Some people view this as an optimization to save scarce code points; others see it as Anglo-Saxon cultural imperialism (and you thought assigning 16-bit values to characters was not highly political?). To make matters worse, a full Japanese dictionary has 50,000 kanji (excluding names), so with only 20,992 code points available for the Han ideographs, choices had to be made. Not all Japanese people think that a consortium of computer companies, even if a few of them are Japanese, is the ideal forum to make these choices.

3

THE DIGITAL LOGIC LEVEL

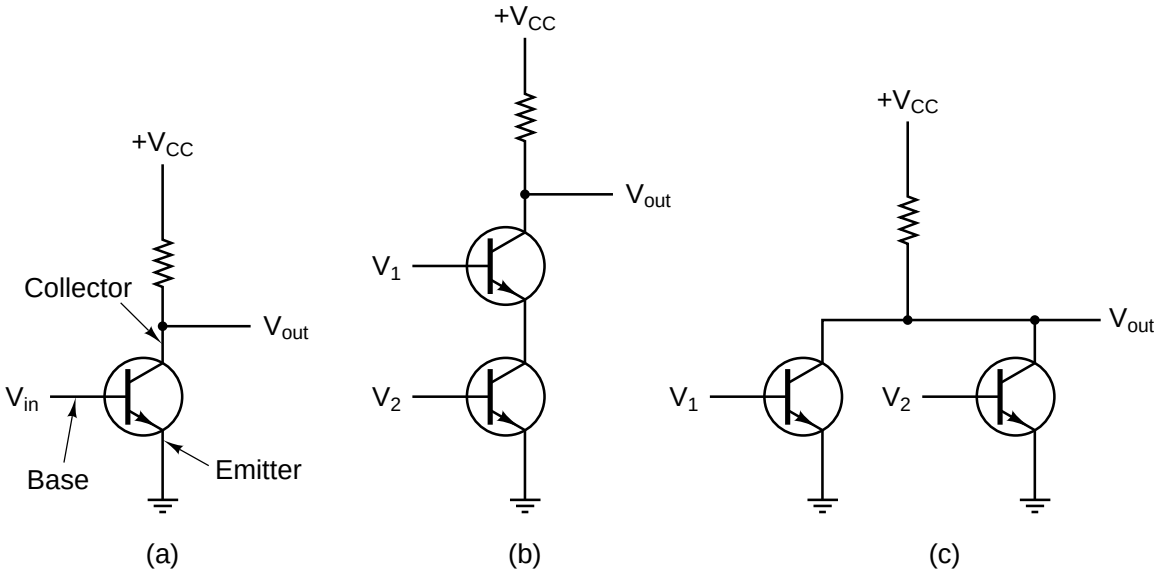
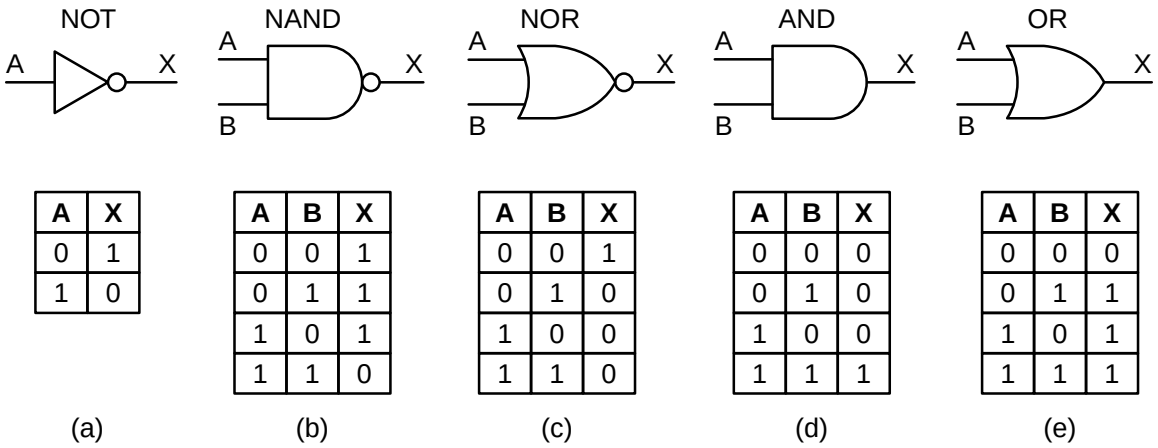


Figure 3-1. (a) A transistor inverter. (b) A NAND gate. (c) A NOR gate.



A	X
0	1
1	0

(a)

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

(b)

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

(c)

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

(d)

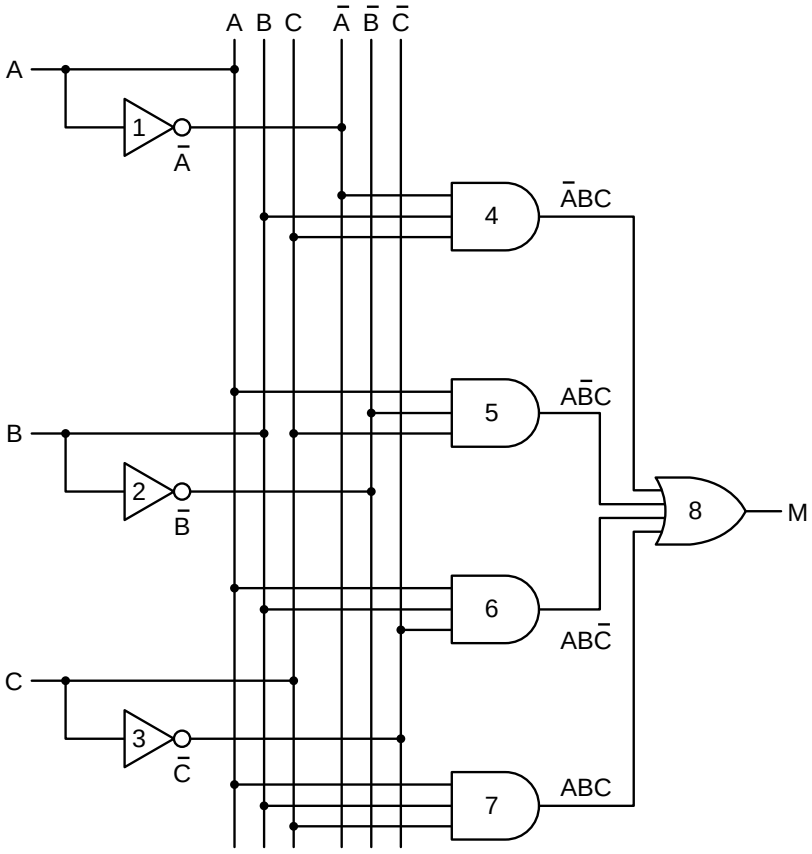
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

(e)

Figure 3-2. The symbols and functional behavior for the five basic gates.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b)

Figure 3-3. (a) The truth table for the majority function of three variables. (b) A circuit for (a).

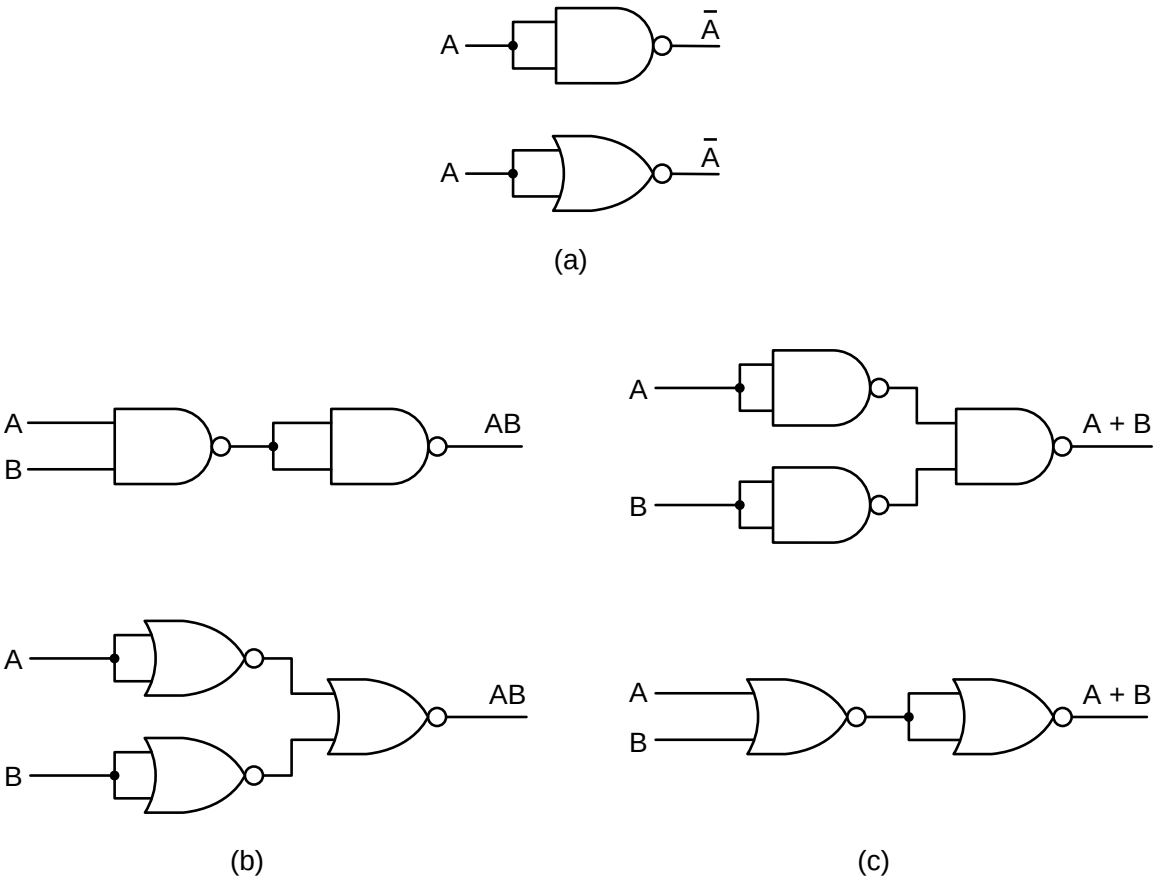
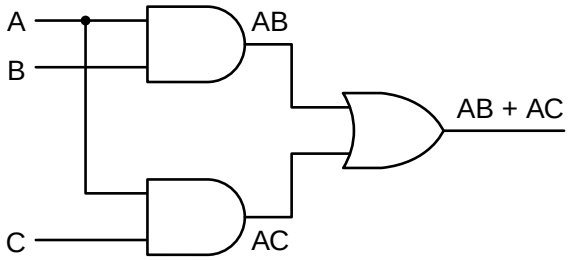
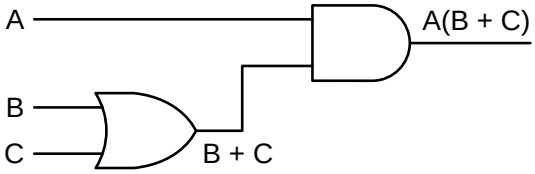


Figure 3-4. Construction of (a) NOT, (b) AND, and (c) OR gates using only NAND gates or only NOR gates.



A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

(a)



A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

(b)

Figure 3-5. Two equivalent functions. (a) $AB + AC$. (b) $A(B + C)$.

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$

Figure 3-6. Some identities of Boolean algebra.

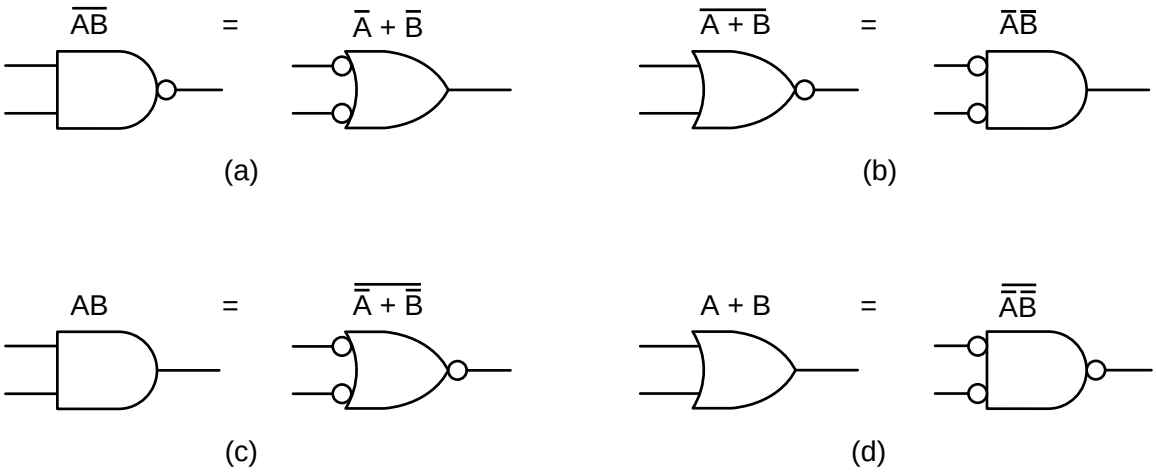
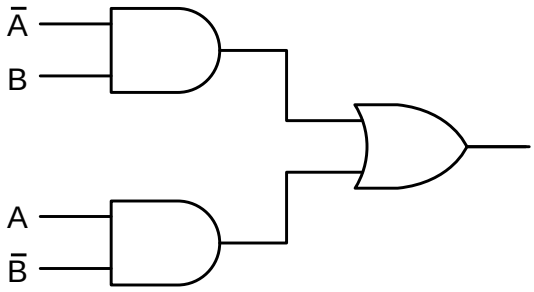


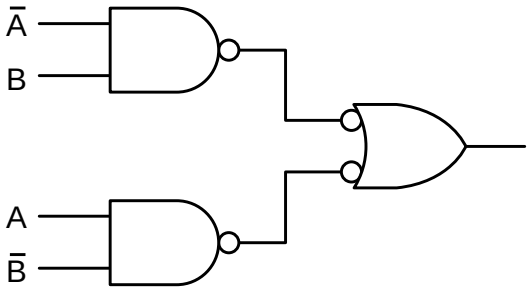
Figure 3-7. Alternative symbols for some gates: (a) NAND. (b) NOR. (c) AND. (d) OR.

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

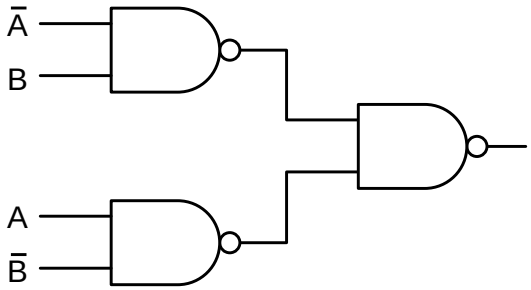
(a)



(b)



(c)



(d)

Figure 3-8. (a) The truth table for the XOR function. (b)-(d) Three circuits for computing it.

A	B	F
0^V	0^V	0^V
0^V	5^V	0^V
5^V	0^V	0^V
5^V	5^V	5^V

(a)

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

(b)

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

(c)

Figure 3-9. (a) Electrical characteristics of a device. (b) Positive logic. (c) Negative logic.

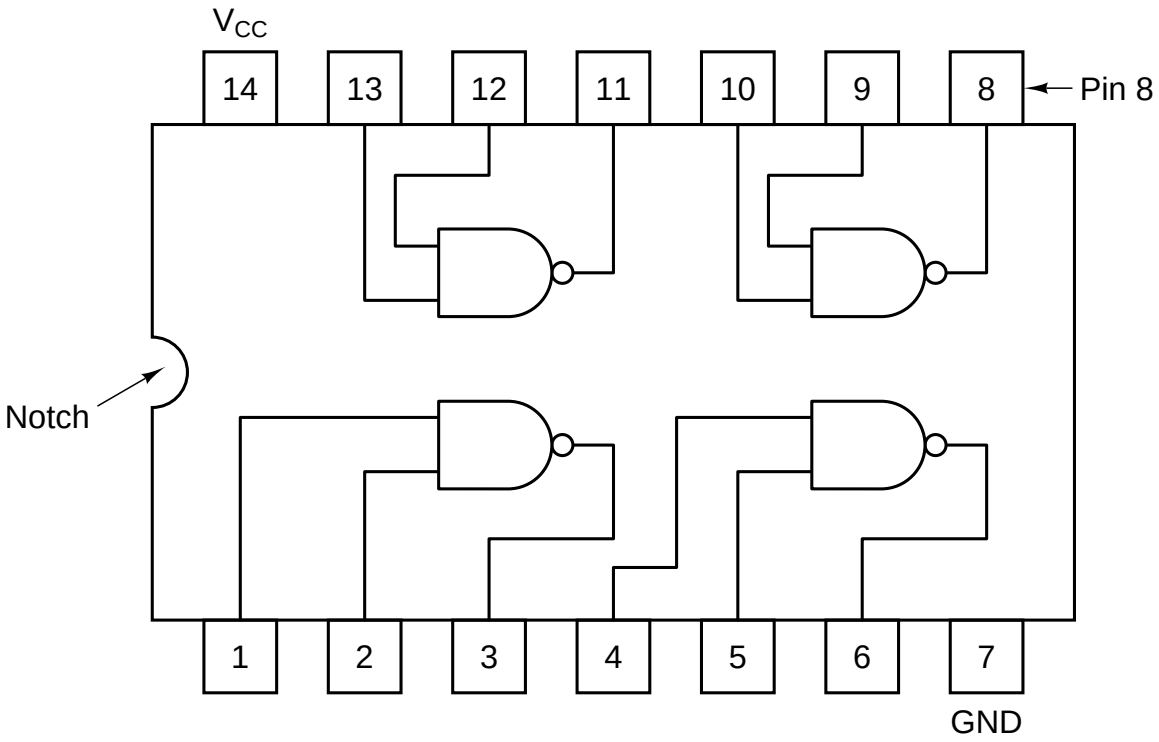


Figure 3-10. An SSI chip containing four gates.

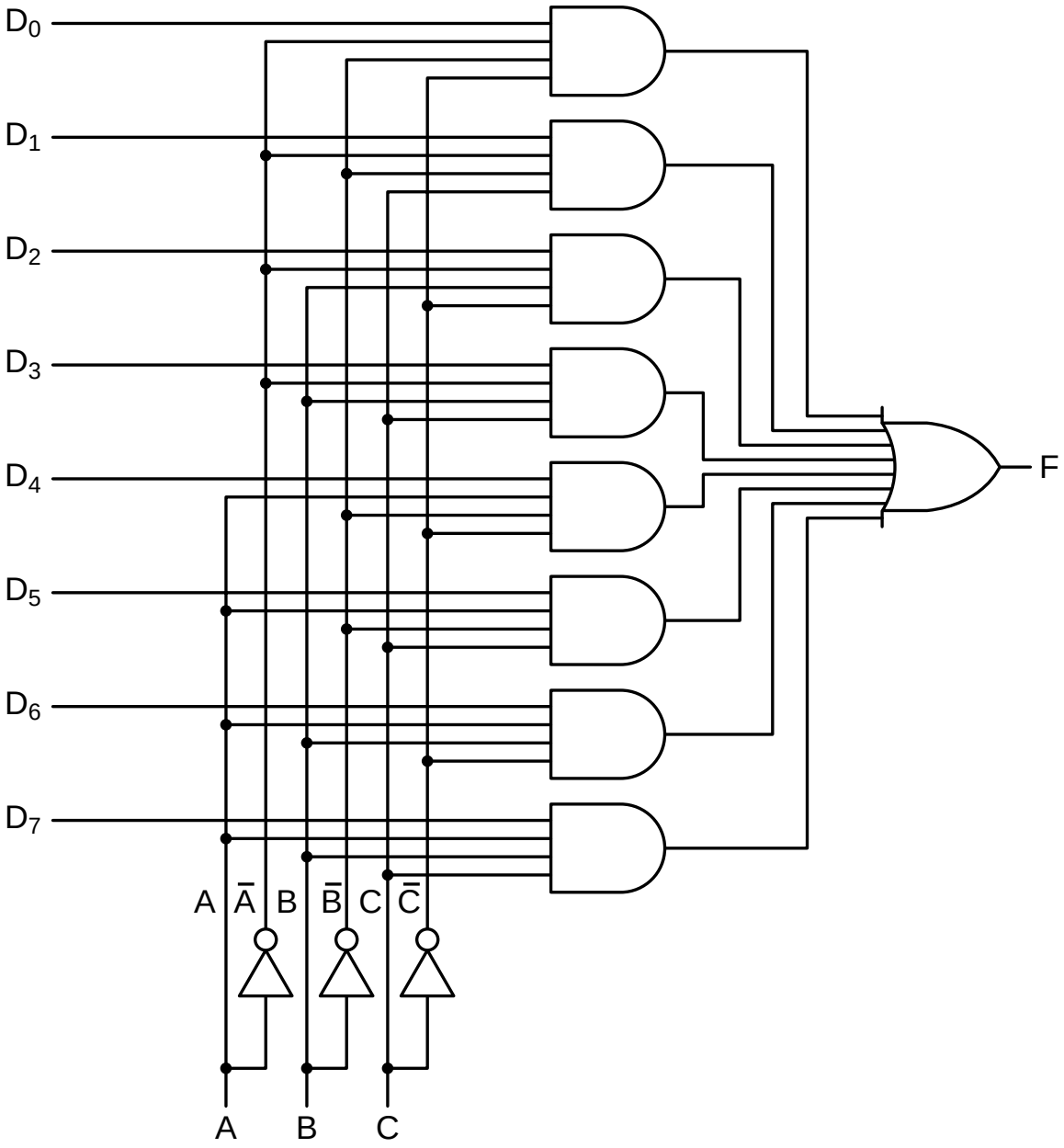


Figure 3-11. An eight-input multiplexer circuit.

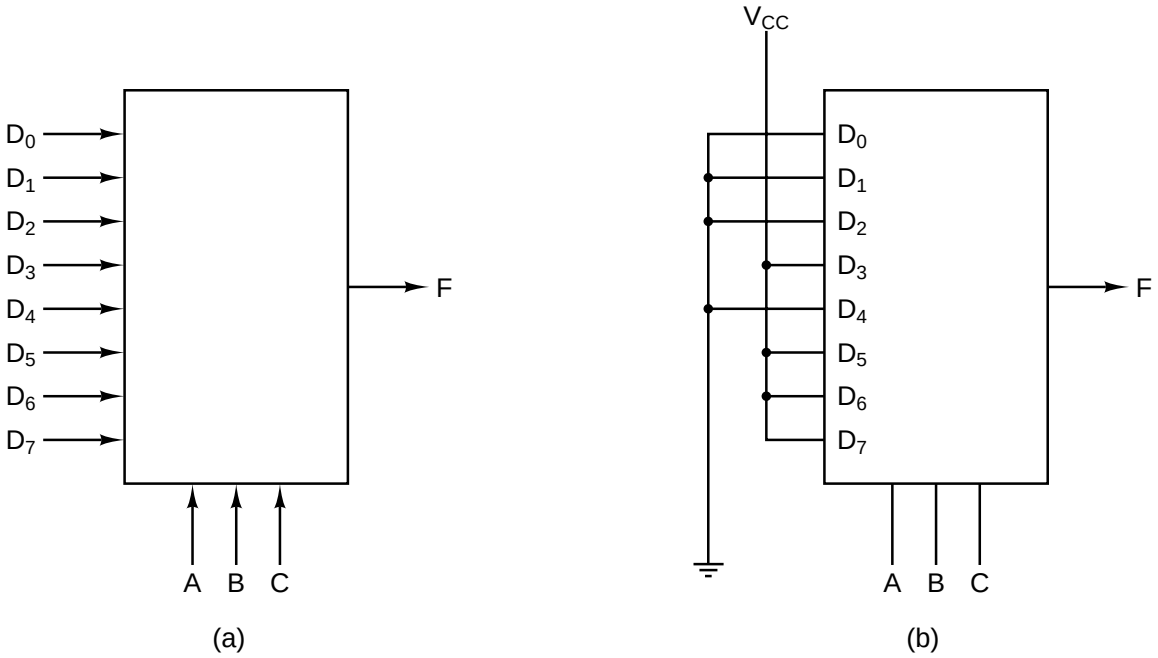


Figure 3-12. (a) An MSI multiplexer.. (b) The same multiplexer wired to compute the majority function.

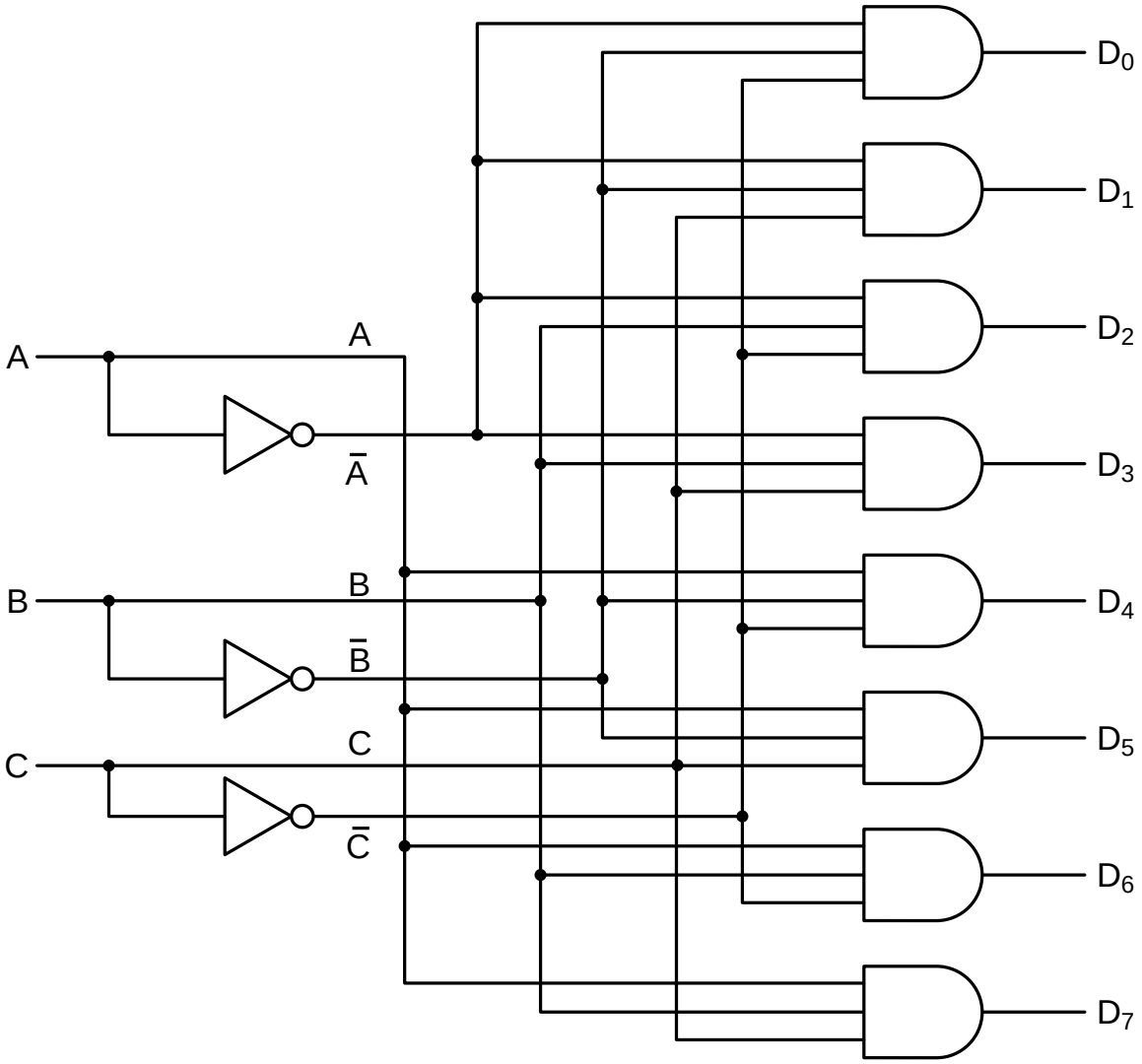


Figure 3-13. A 3-to-8 decoder circuit.

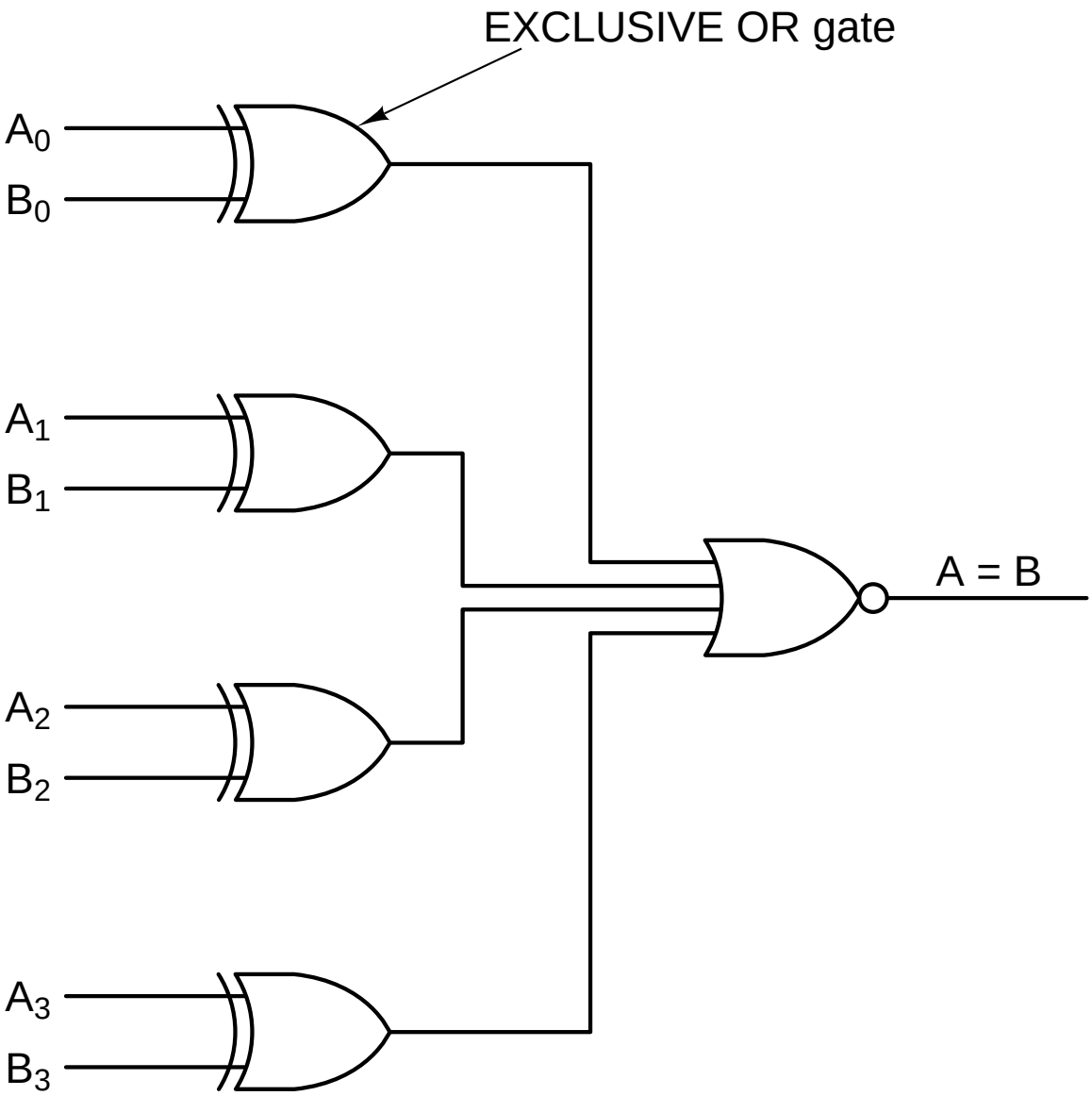


Figure 3-14. A simple 4-bit comparator.

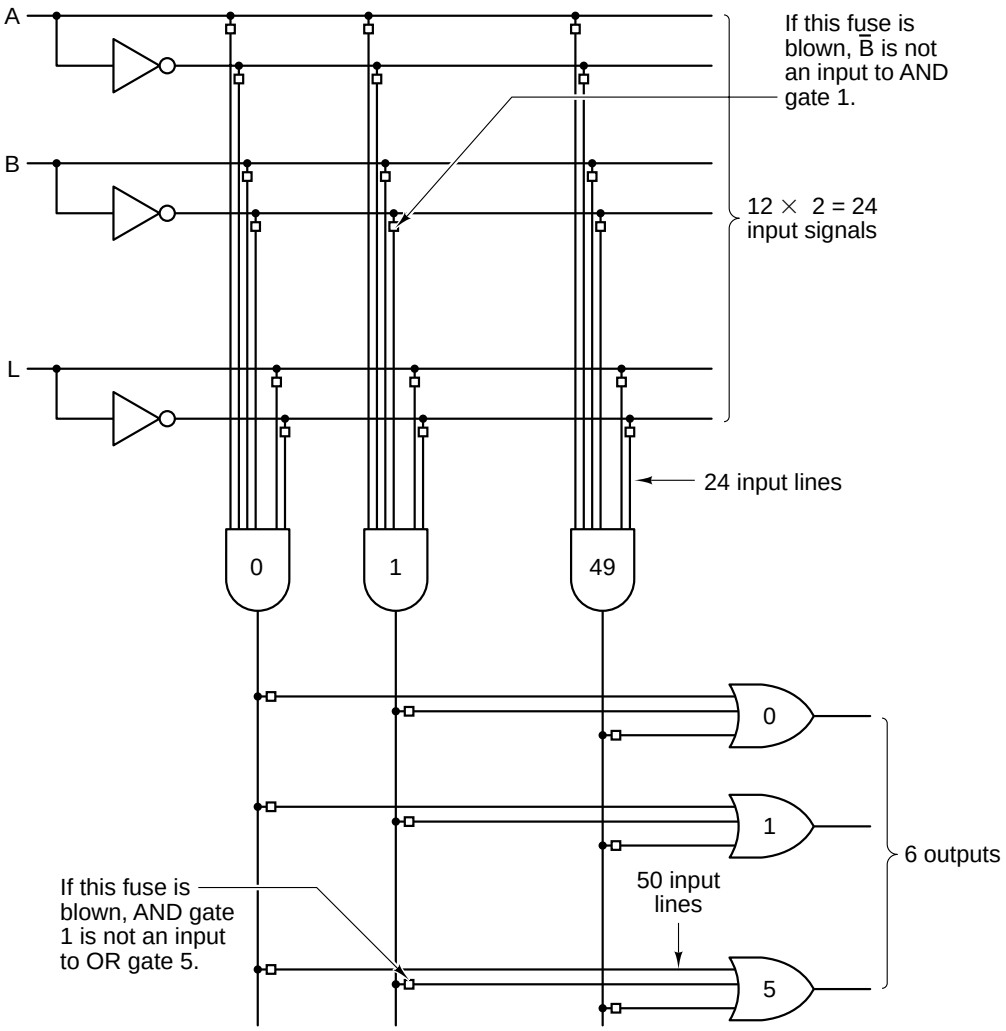


Figure 3-15. A 12-input, 6-output programmable logic array. The little squares represent fuses that can be burned out to determine the function to be computed. The fuses are arranged in two matrices: the upper one for the AND gates and the lower one for the OR gates.

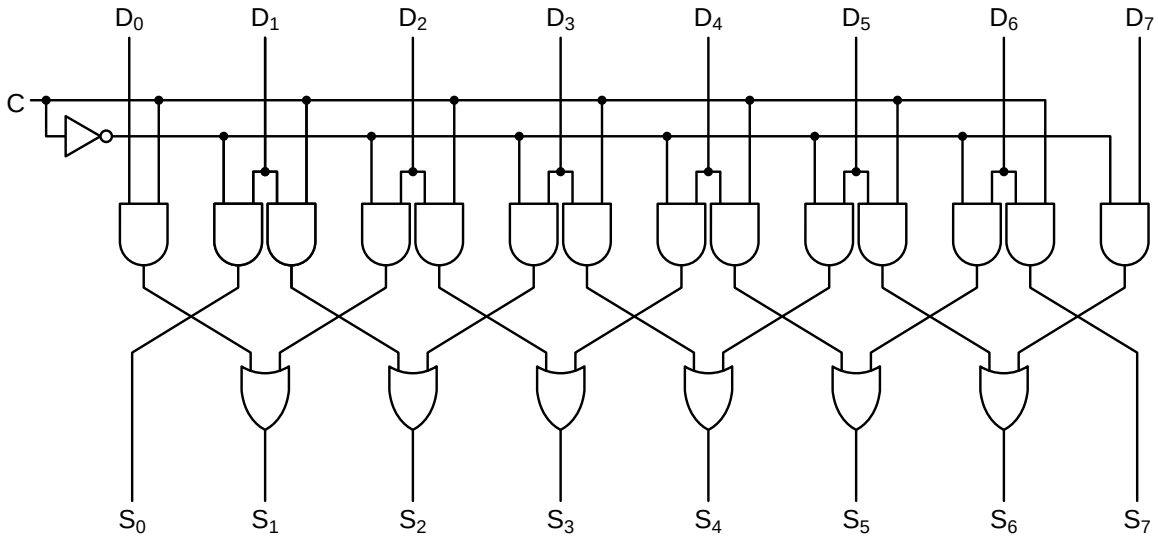


Figure 3-16. A 1-bit left/right shifter.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

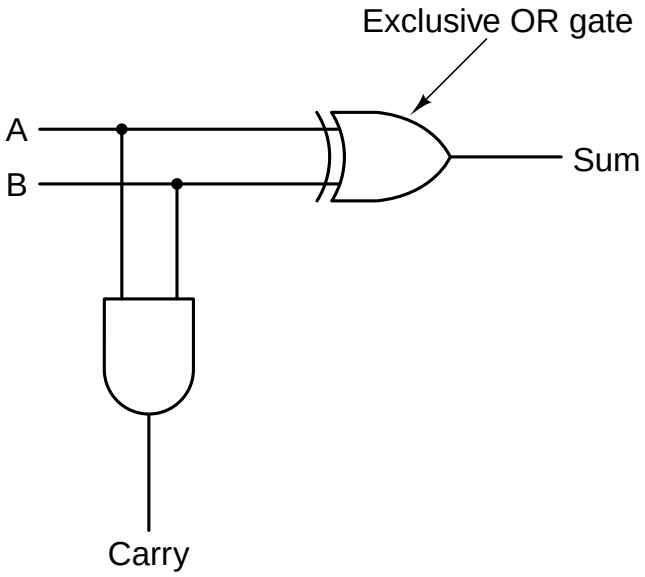
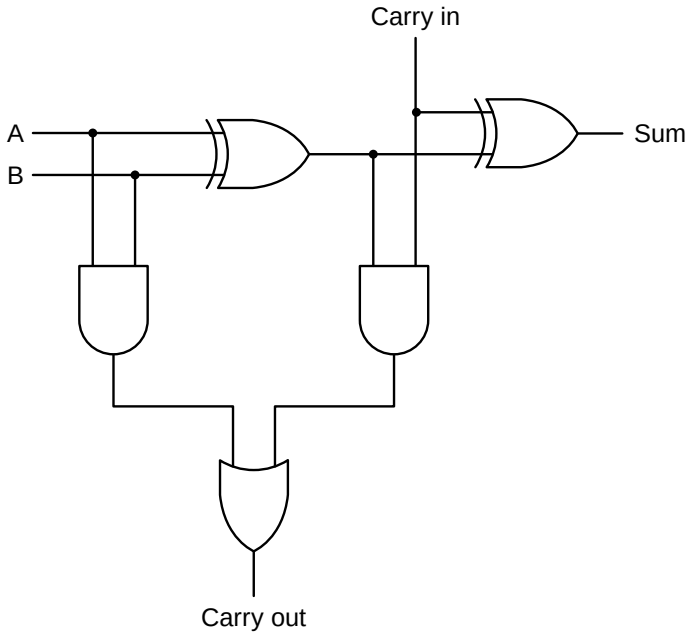


Figure 3-17. (a) Truth table for 1-bit addition. (b) A circuit for a half adder.

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)



(b)

Figure 3-18. (a) Truth table for full adder. (b) Circuit for a full adder.

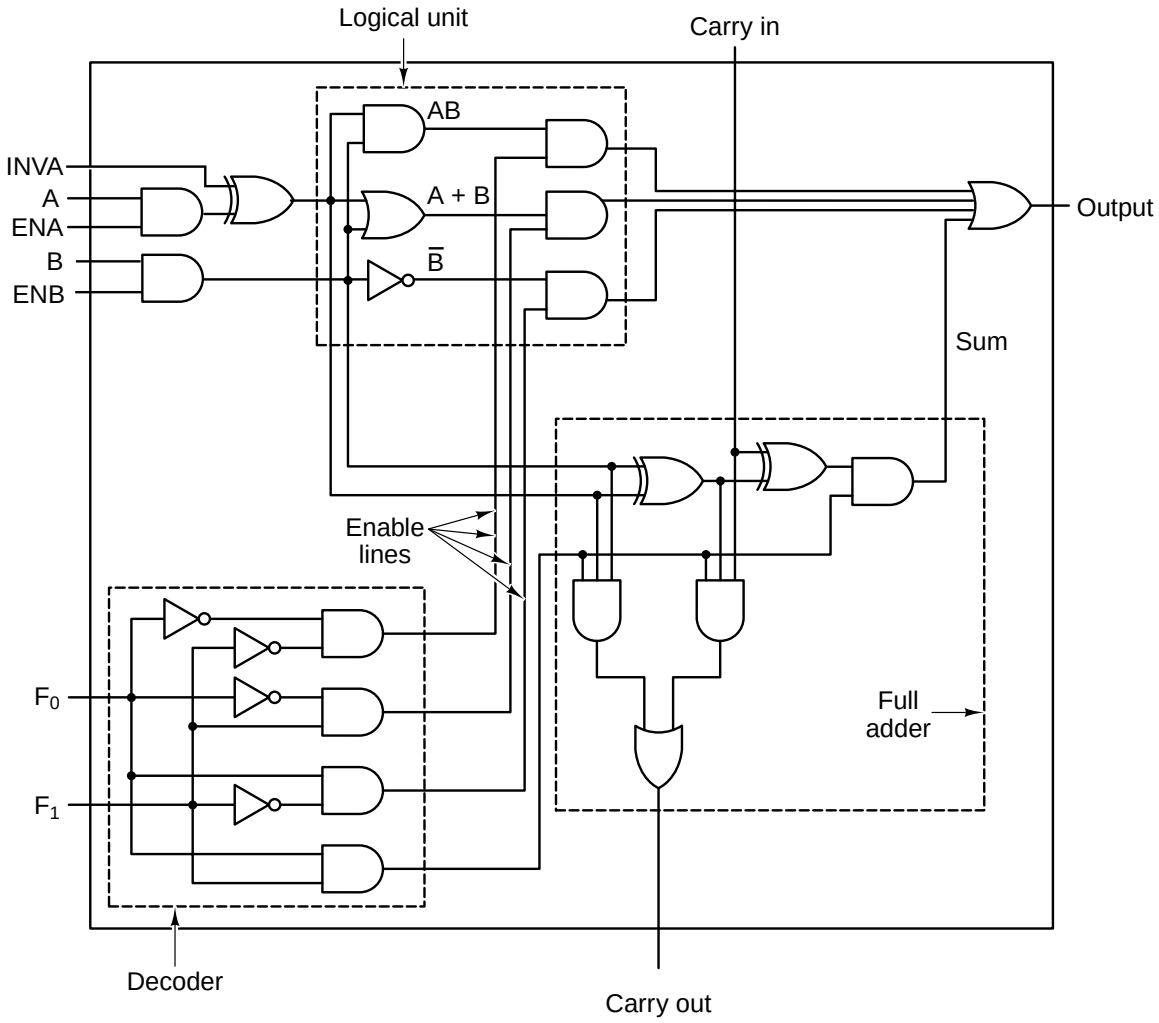


Figure 3-19. A 1-bit ALU.

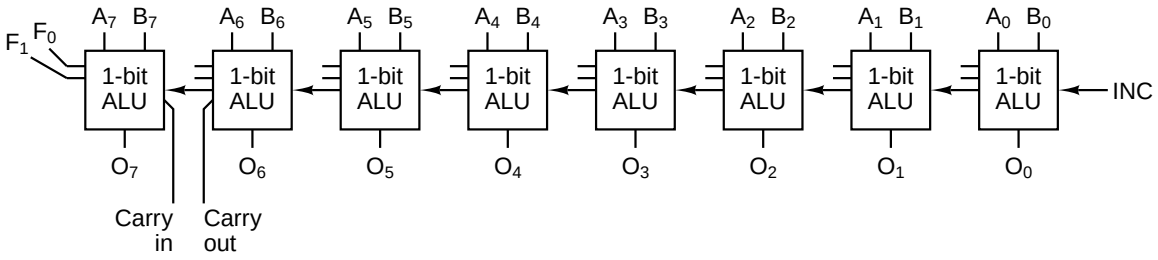


Figure 3-20. Eight 1-bit ALU slices connected to make an 8-bit ALU. The enables and invert signals are not shown for simplicity.

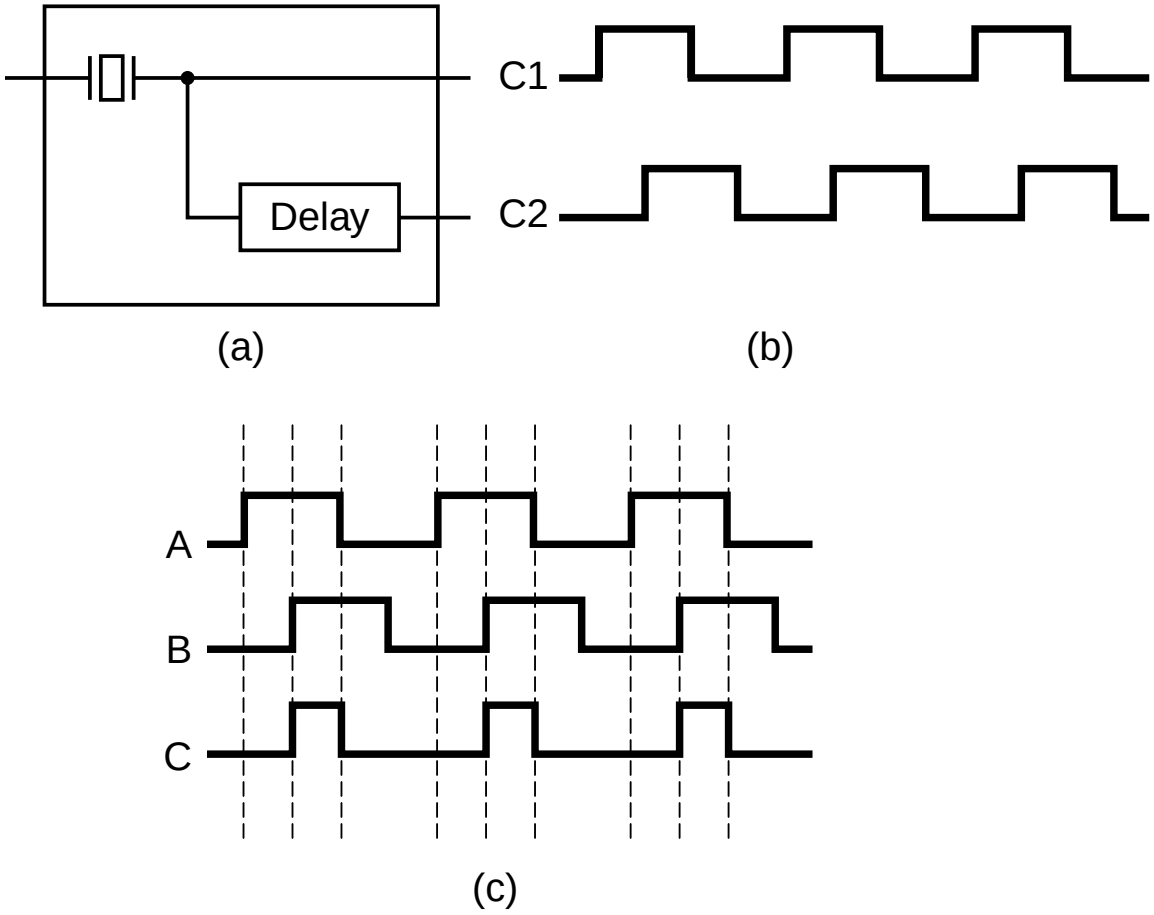
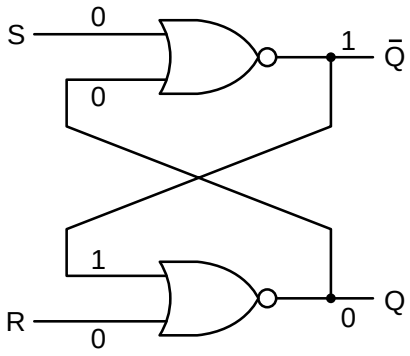
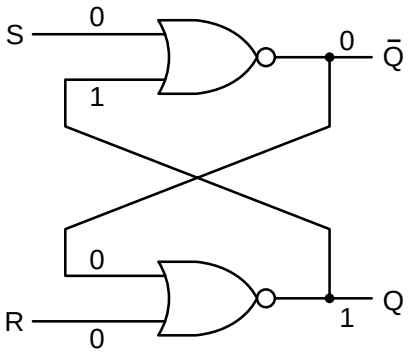


Figure 3-21. (a) A clock. (b) The timing diagram for the clock. (c) Generation of an asymmetric clock.



(a)



(b)

A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

(c)

Figure 3-22. (a) NOR latch in state 0. (b) NOR latch in state 1. (c) Truth table for NOR.

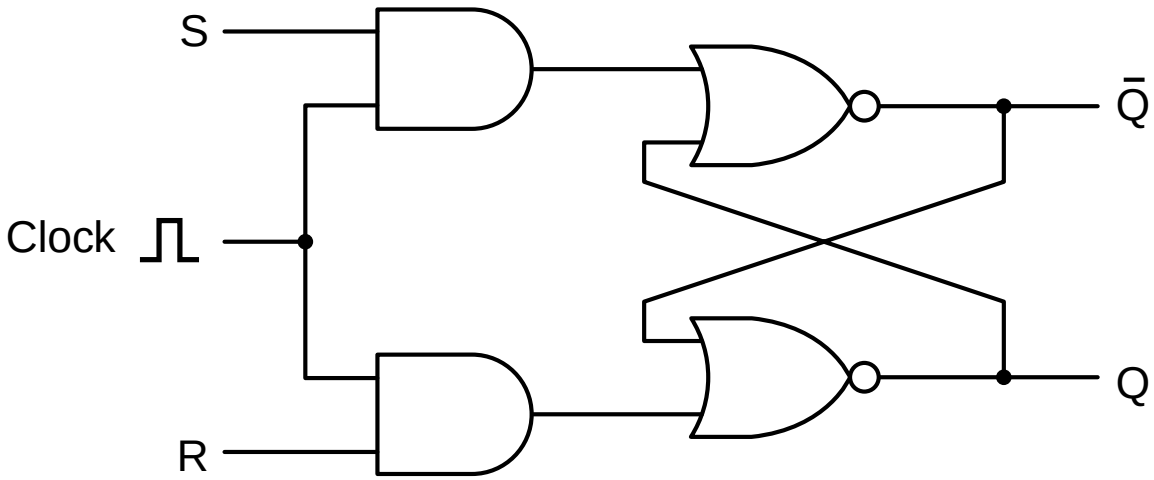


Figure 3-23. A clocked SR latch.

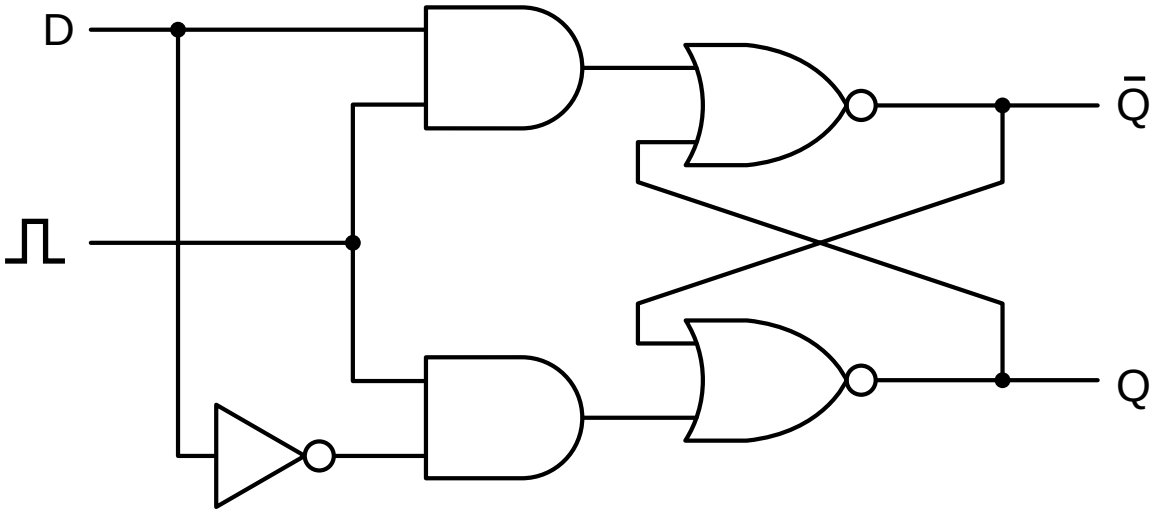


Figure 3-24. A clocked D latch.

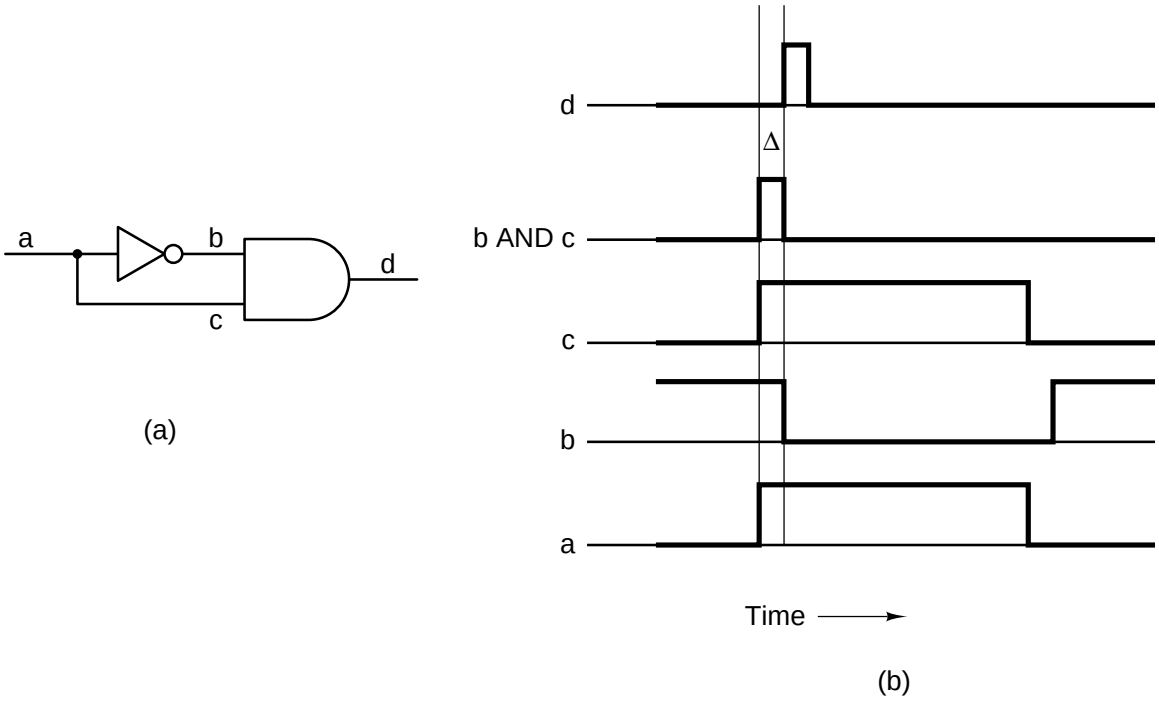


Figure 3-25. (a) A pulse generator. (b) Timing at four points in the circuit.

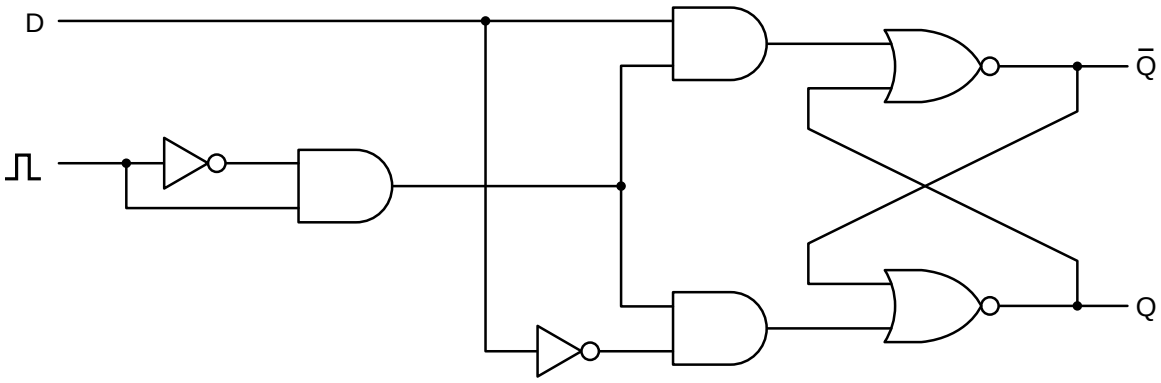


Figure 3-26. A D flip-flop.

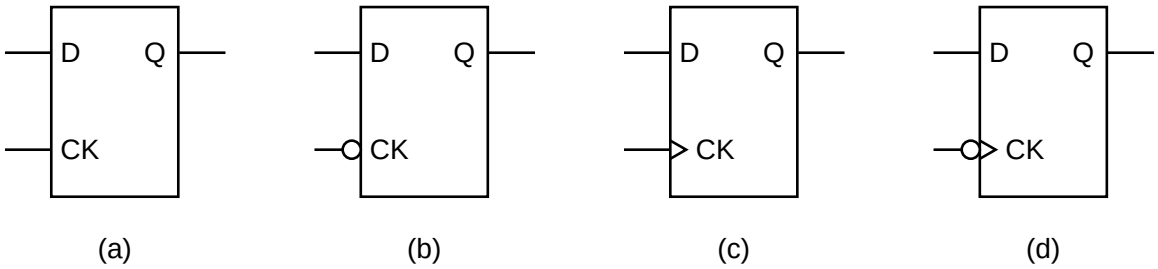


Figure 3-27. D latches and flip-flops.

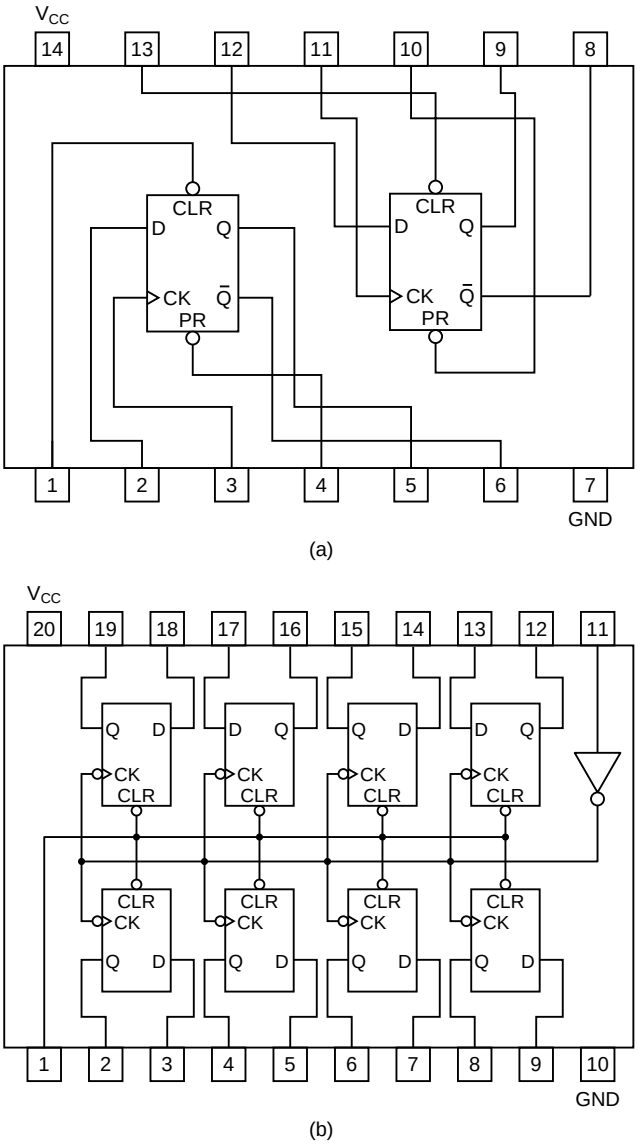


Figure 3-28. (a) Dual D flip-flop. (b) Octal flip-flop.

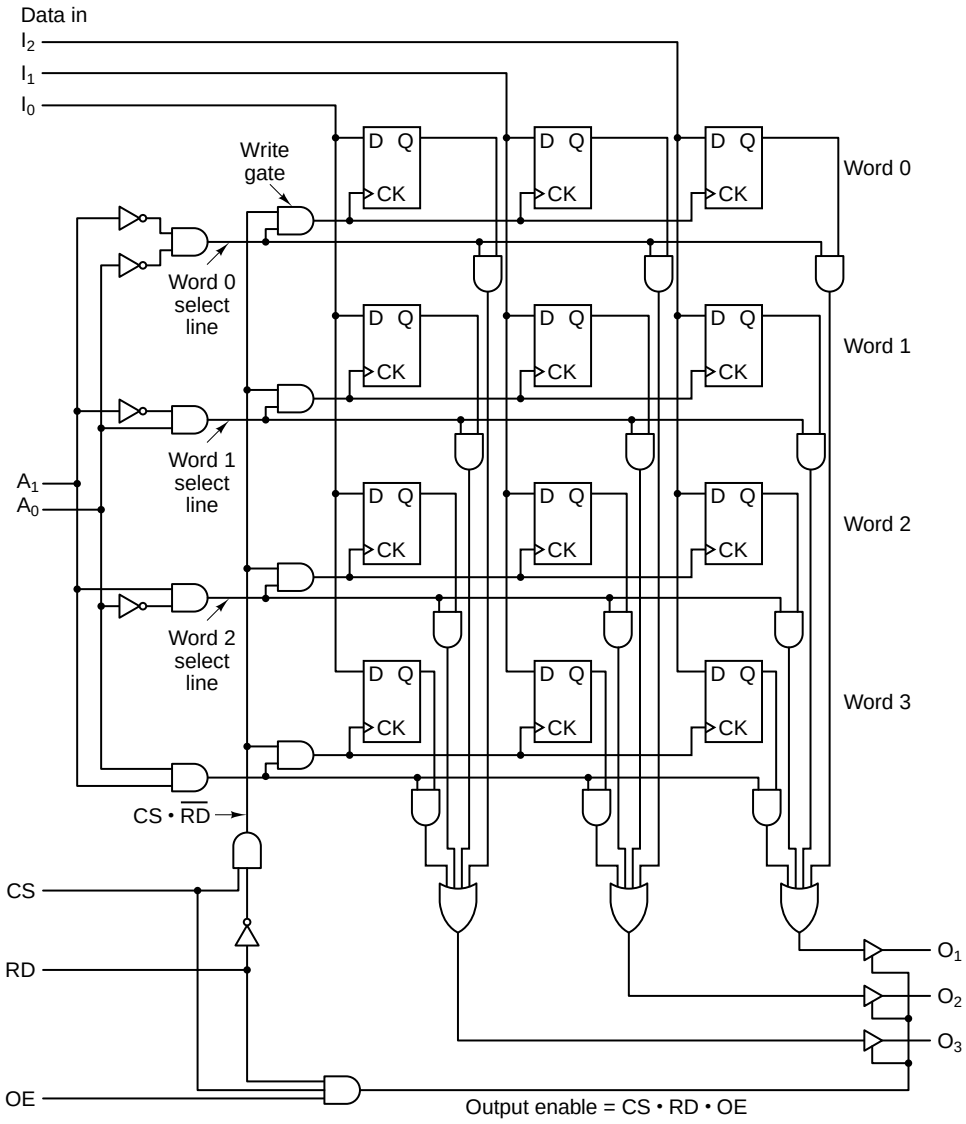


Figure 3-29. Logic diagram for a 4×3 memory. Each row is one of the four 3-bit words. A read or write operation always reads or writes a complete word.

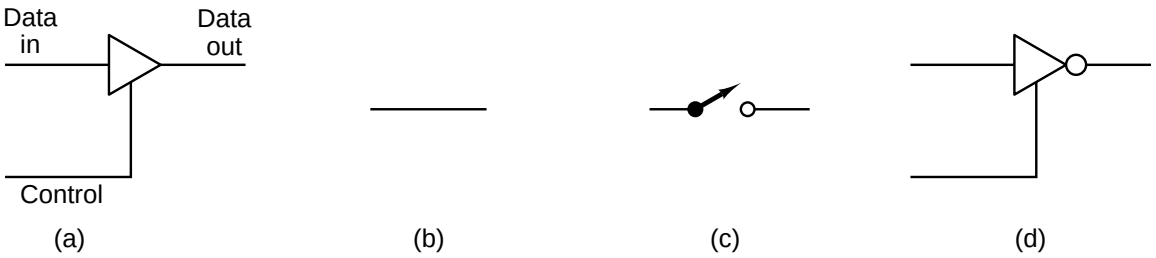


Figure 3-30. (a) A noninverting buffer. (b) Effect of (a) when control is high. (c) Effect of (a) when control is low. (d) An inverting buffer.

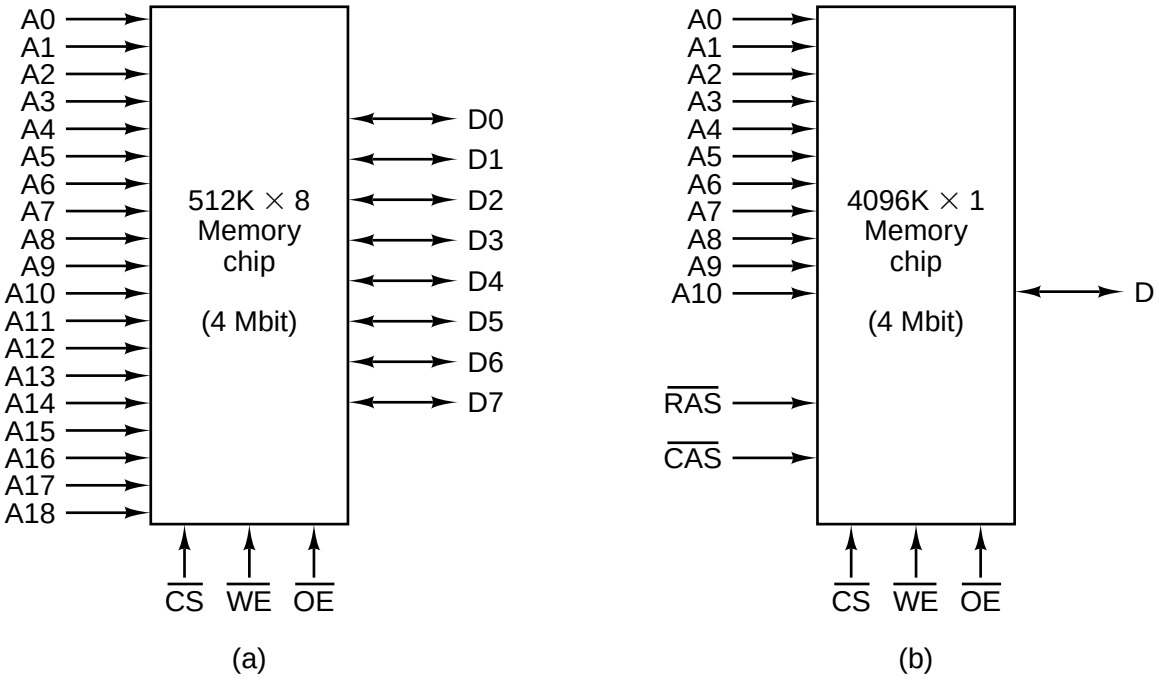


Figure 3-31. Two ways of organizing a 4-Mbit memory chip.

Type	Category	Erasure	Byte alterable	Volatile	Typical use
SRAM	Read/write	Electrical	Yes	Yes	Level 2 cache
DRAM	Read/write	Electrical	Yes	Yes	Main memory
ROM	Read-only	Not possible	No	No	Large volume appliances
PROM	Read-only	Not possible	No	No	Small volume equipment
EPROM	Read-mostly	UV light	No	No	Device prototyping
EEPROM	Read-mostly	Electrical	Yes	No	Device prototyping
Flash	Read/write	Electrical	No	No	Film for digital camera

Figure 3-32. A comparison of various memory types.

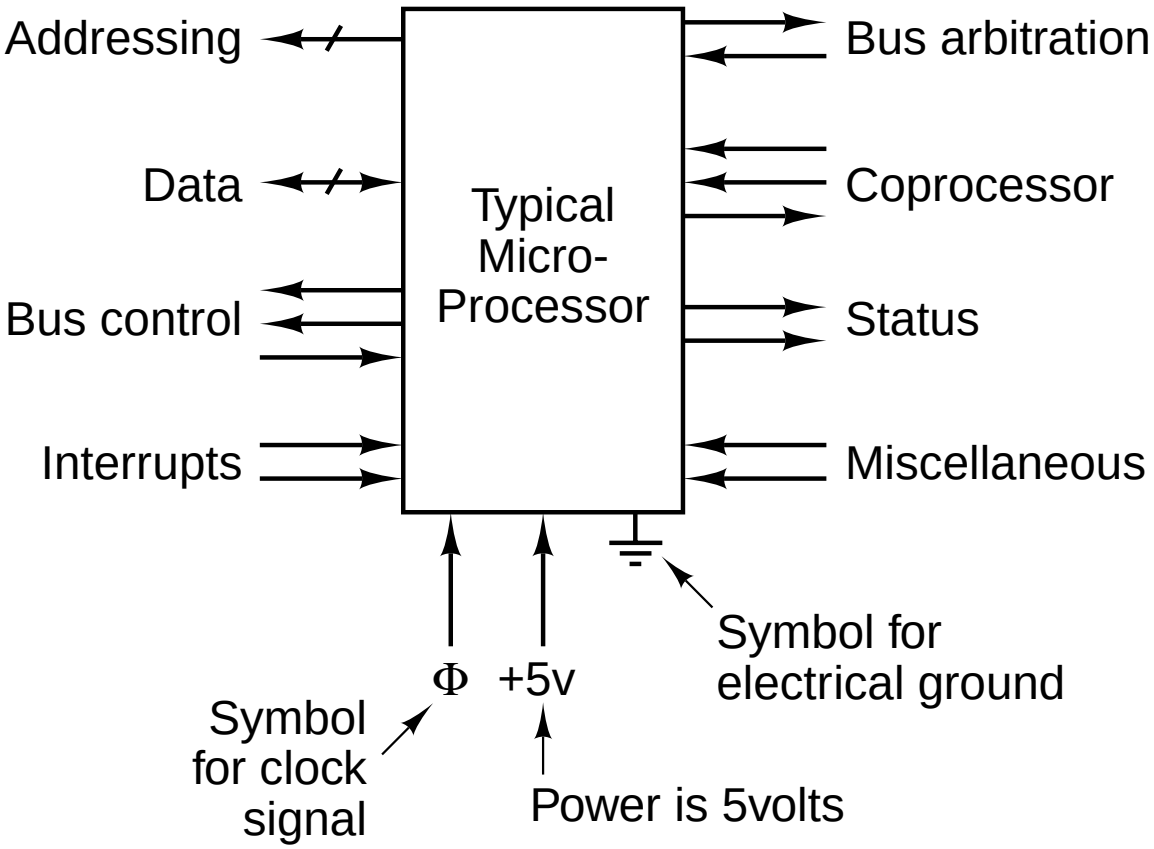


Figure 3-33. The logical pinout of a generic CPU. The arrows indicate input signals and output signals. The short diagonal lines indicate that multiple pins are used. For a specific CPU, a number will be given to tell how many.

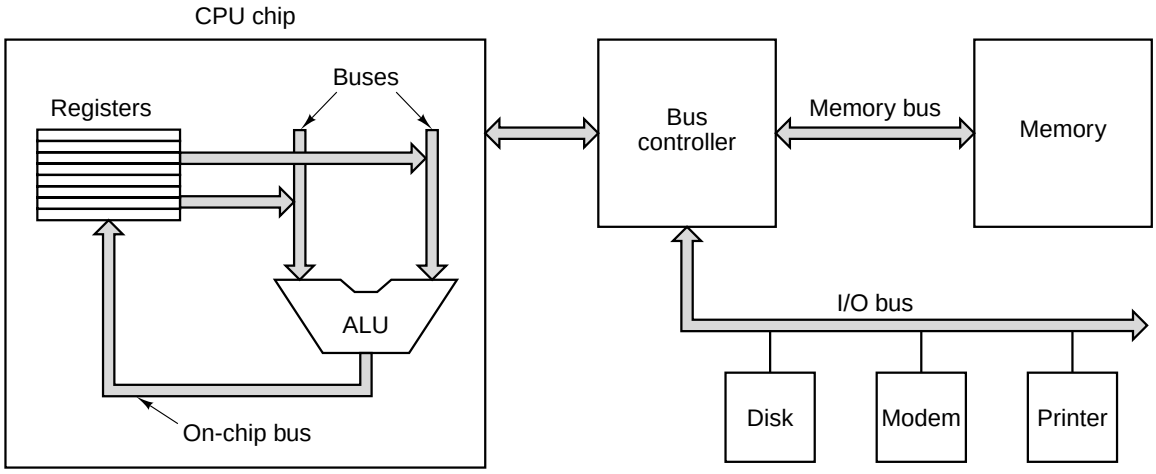


Figure 3-34. A computer system with multiple buses.

Master	Slave	Example
CPU	Memory	Fetching instructions and data
CPU	I/O device	Initiating data transfer
CPU	Coprocessor	CPU handing instruction off to coprocessor
I/O	Memory	DMA (Direct Memory Access)
Coprocessor	CPU	Coprocessor fetching operands from CPU

Figure 3-35. Examples of bus masters and slaves.

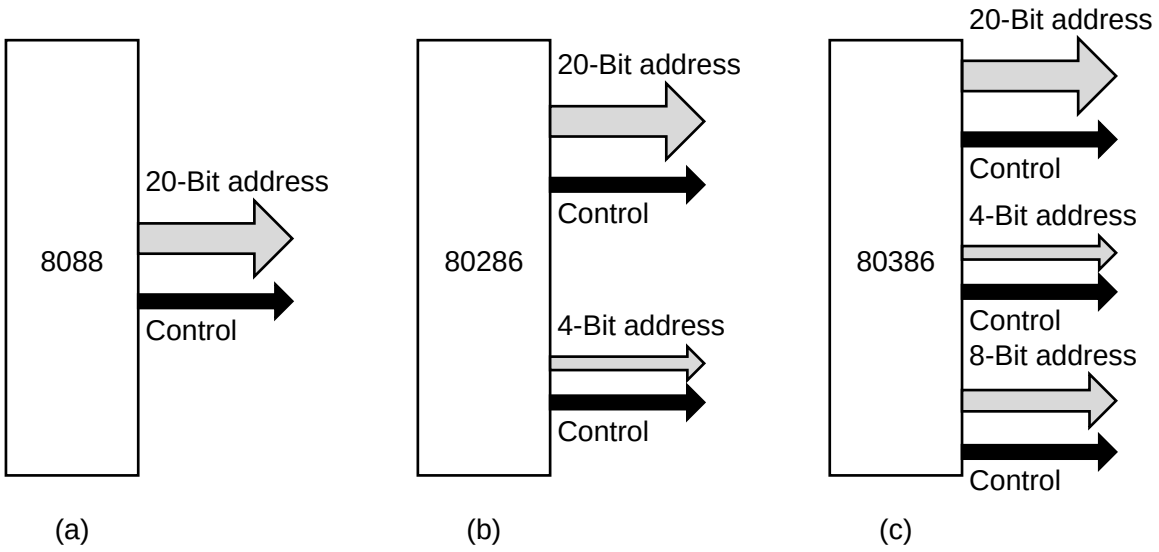
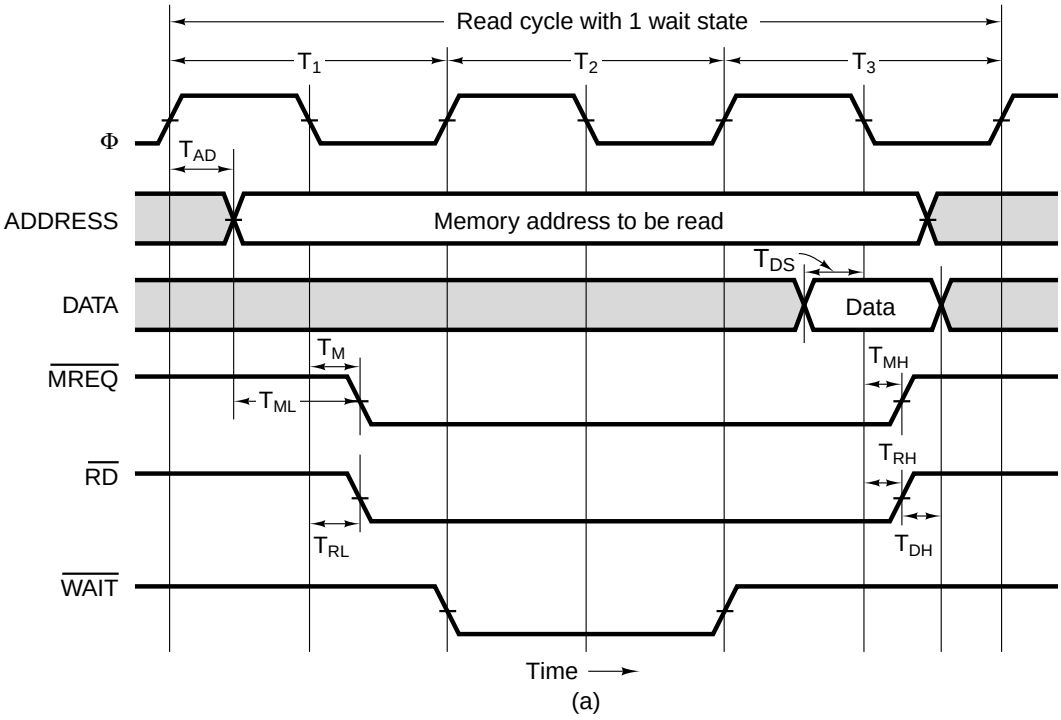


Figure 3-36. Growth of an address bus over time.



Symbol	Parameter	Min	Max	Unit
T_{AD}	Address output delay		11	nsec
T_{ML}	Address stable prior to $\overline{MREQ}$	6		nsec
T_M	$\overline{MREQ}$ delay from falling edge of Φ in T_1		8	nsec
T_{RL}	RD delay from falling edge of Φ in T_1		8	nsec
T_{DS}	Data setup time prior to falling edge of Φ	5		nsec
T_{MH}	$\overline{MREQ}$ delay from falling edge of Φ in T_3		8	nsec
T_{RH}	$\overline{RD}$ delay from falling edge of Φ in T_3		8	nsec
T_{DH}	Data hold time from negation of $\overline{RD}$	0		nsec

(b)

Figure 3-37. (a) Read timing on a synchronous bus. (b) Specification of some critical times.

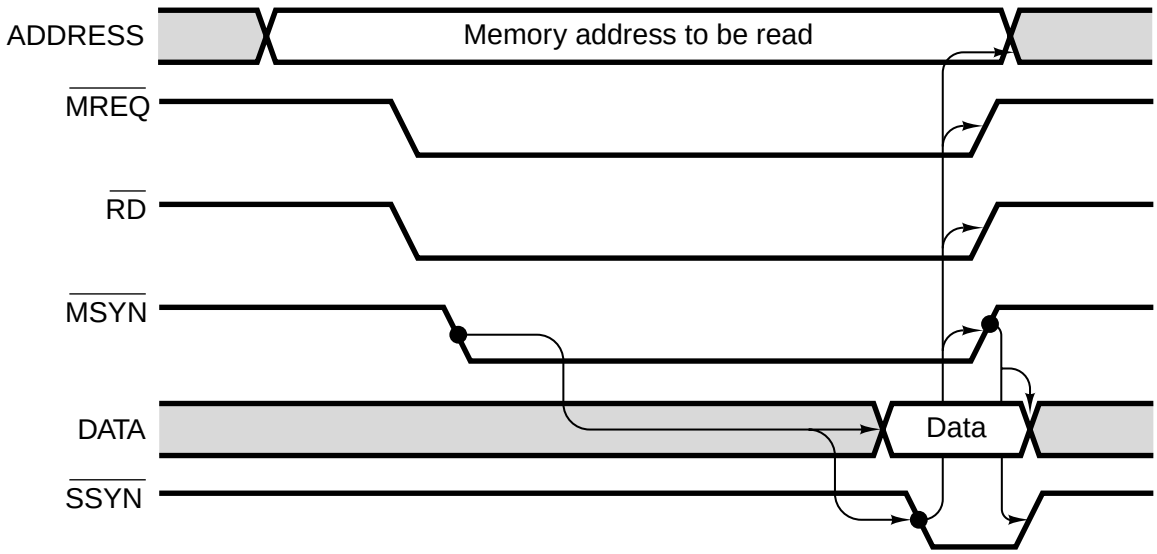


Figure 3-38. Operation of an asynchronous bus.

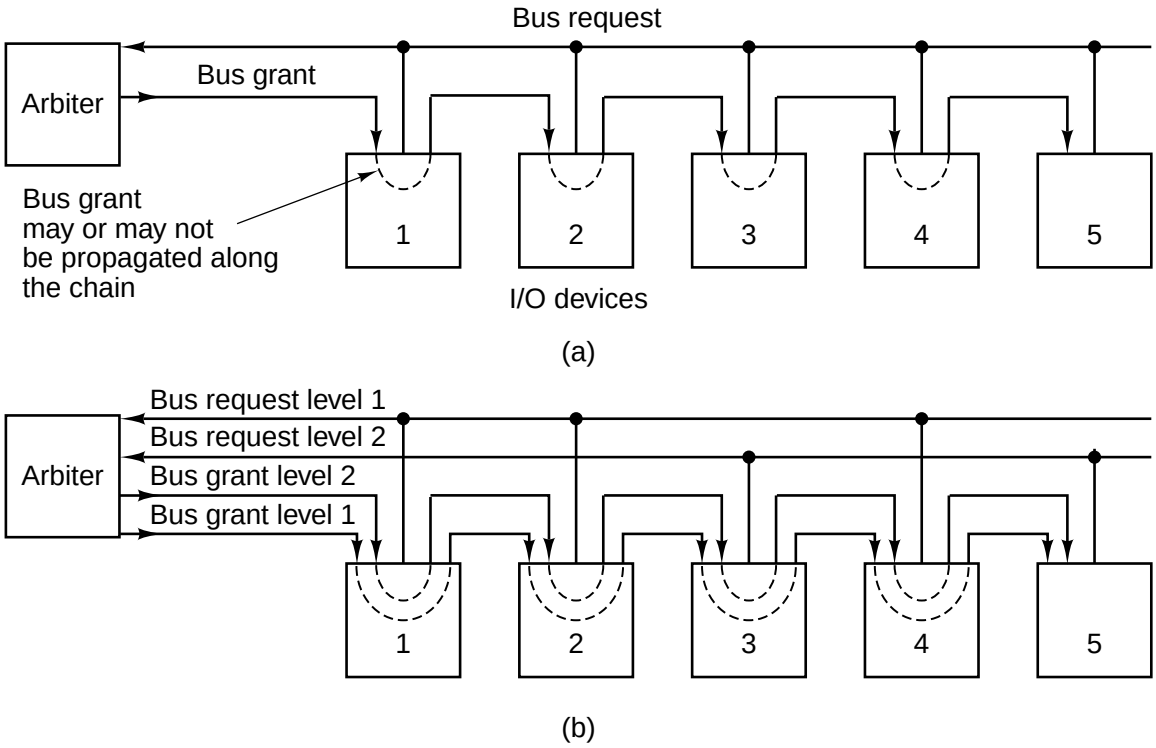


Figure 3-39. (a) A centralized one-level bus arbiter using daisy chaining. (b) The same arbiter, but with two levels.

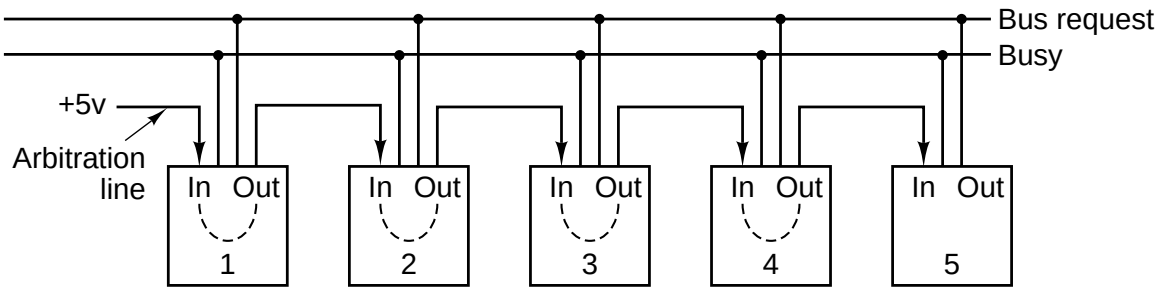


Figure 3-40. Decentralized bus arbitration.

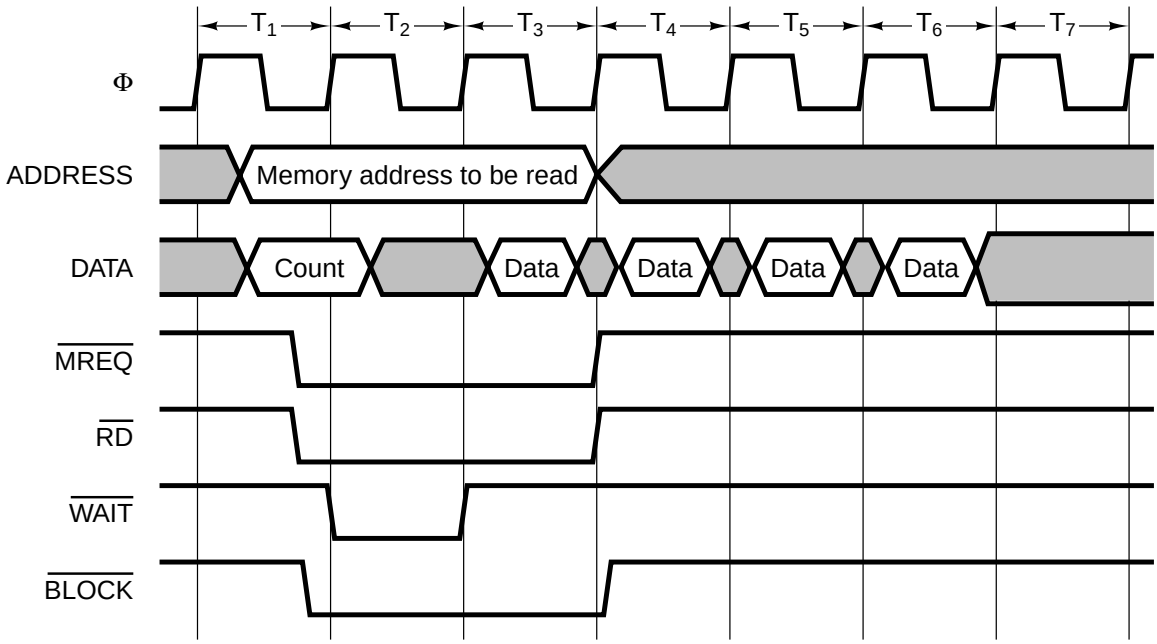


Figure 3-41. A block transfer.

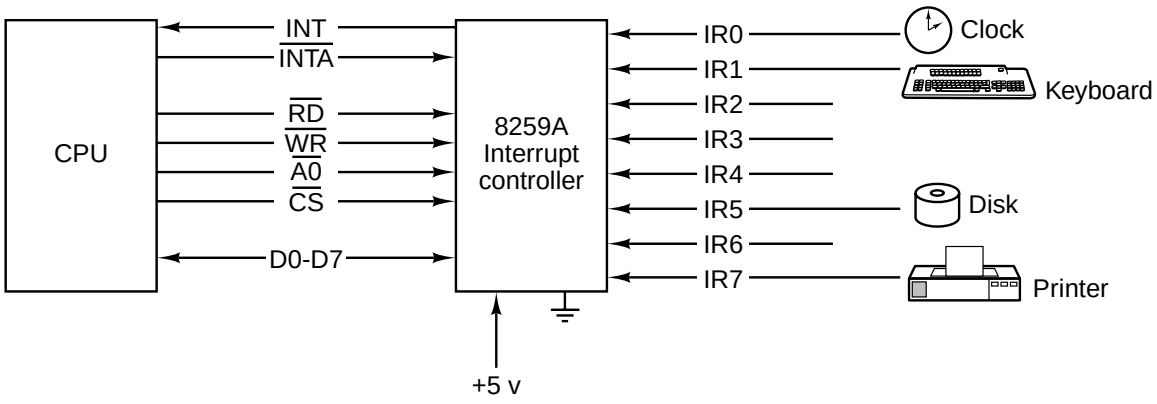


Figure 3-42. Use of the 8259A interrupt controller.

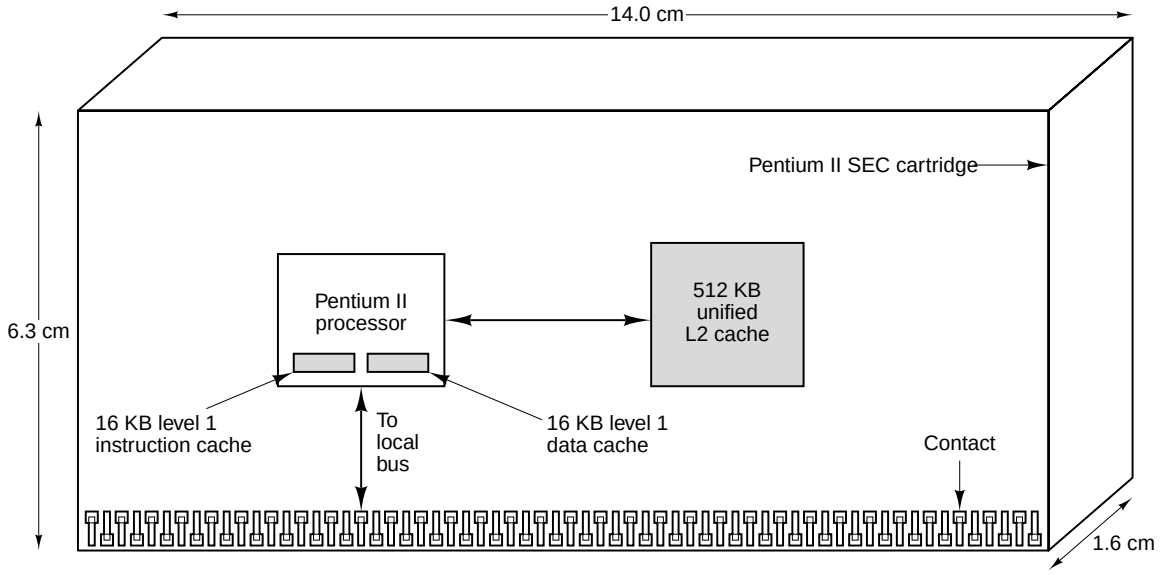


Figure 3-43. The Pentium II SEC package.

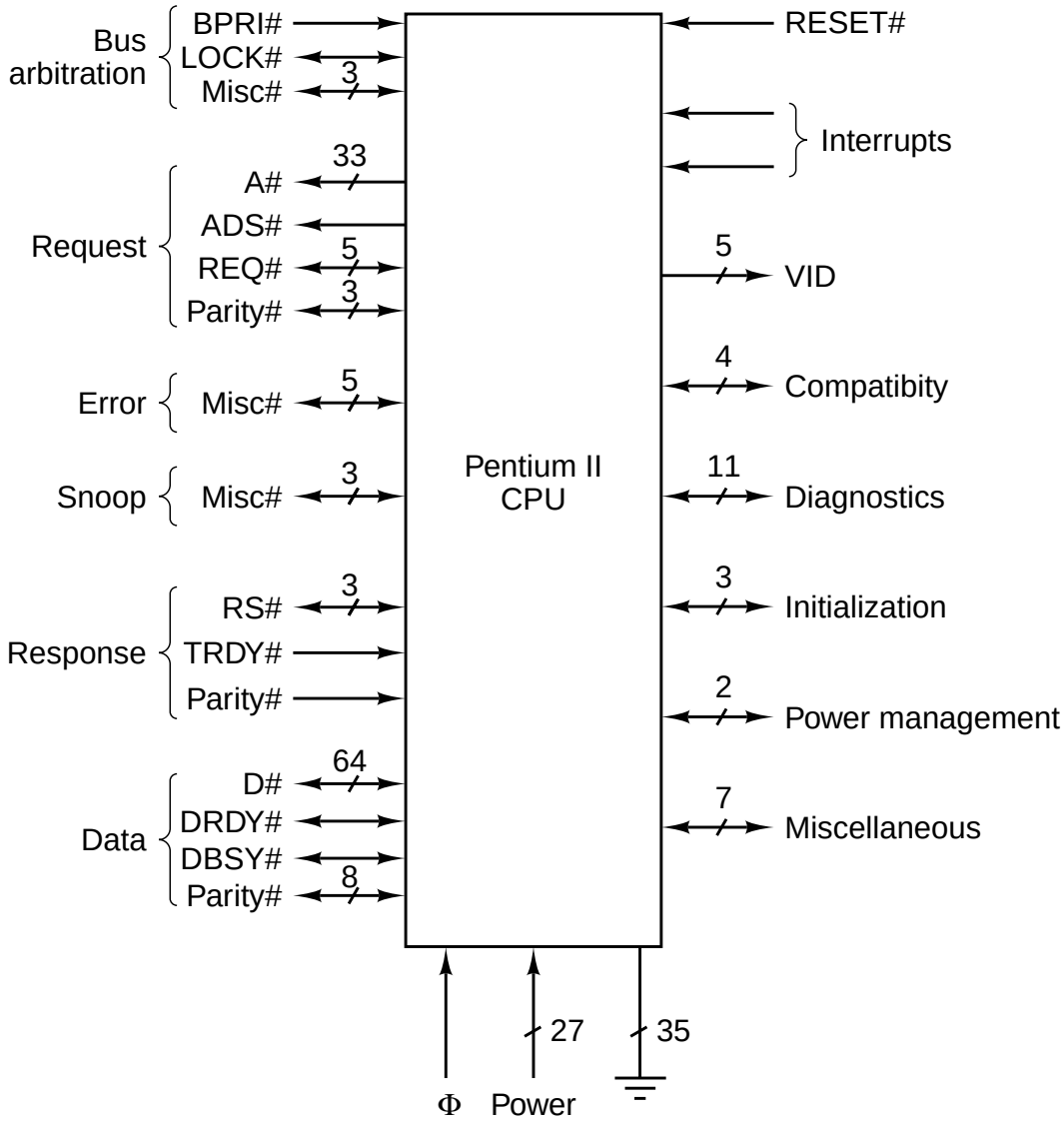


Figure 3-44. Logical pinout of the Pentium II. Names in upper case are the official Intel names for individual signals. Names in mixed case are groups of related signals or signal descriptions.

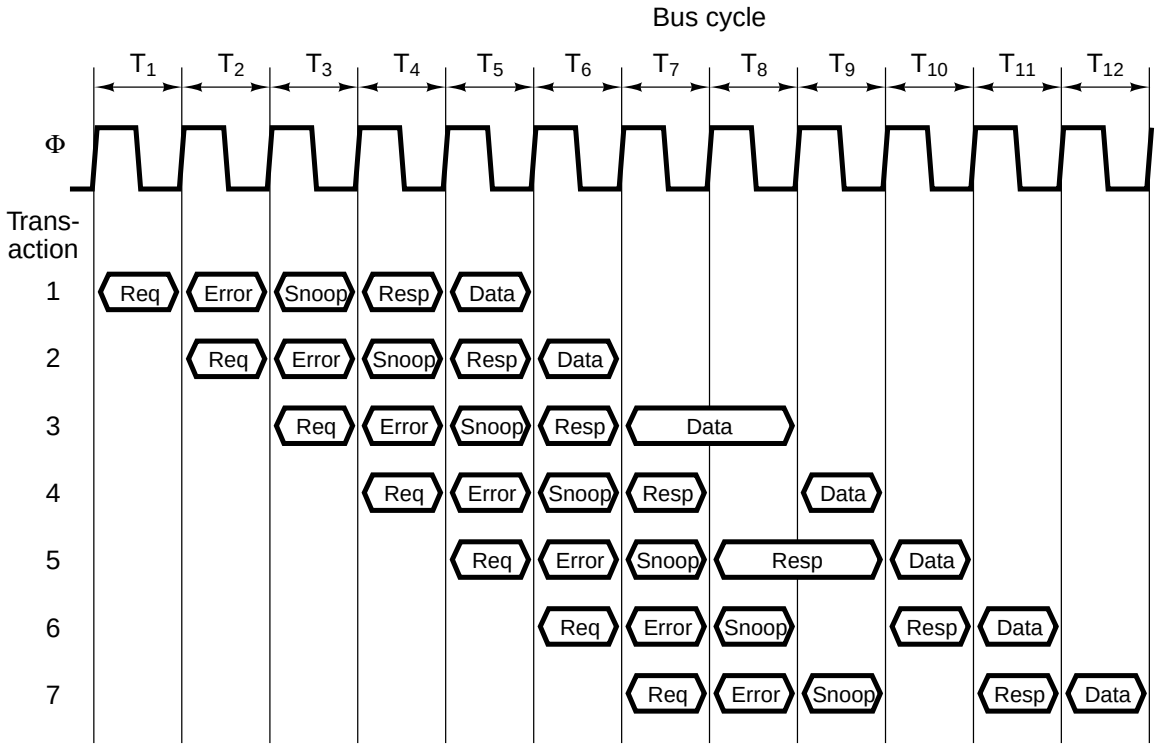


Figure 3-45. Pipelining requests on the Pentium II's memory bus.

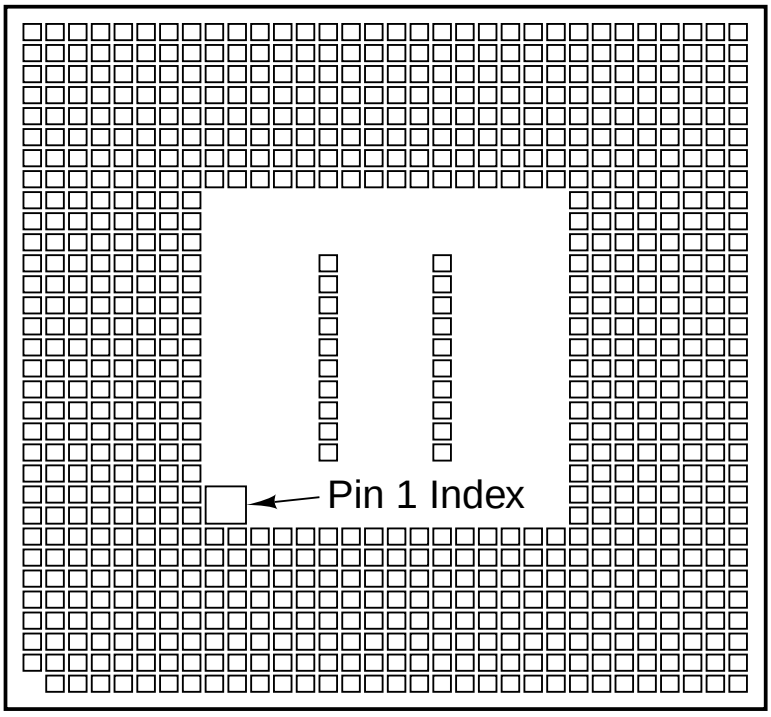


Figure 3-46. The UltraSPARC II CPU chip.

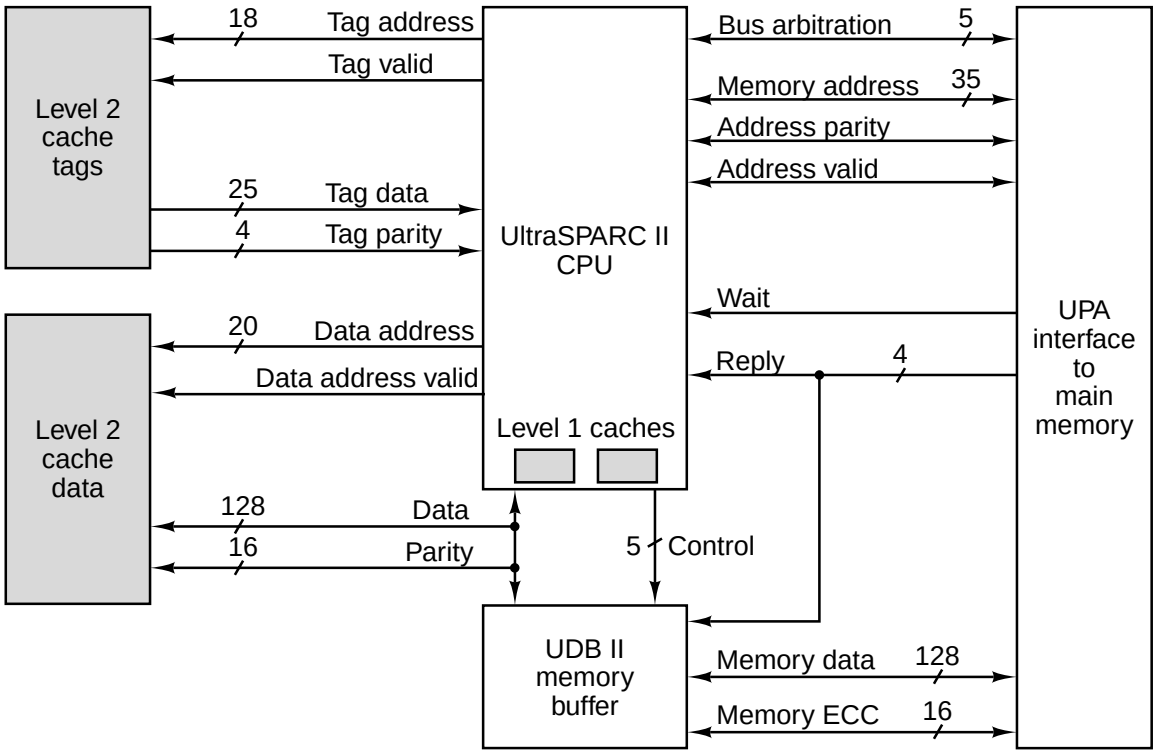


Figure 3-47. The main features of the core of an UltraSPARC II system.

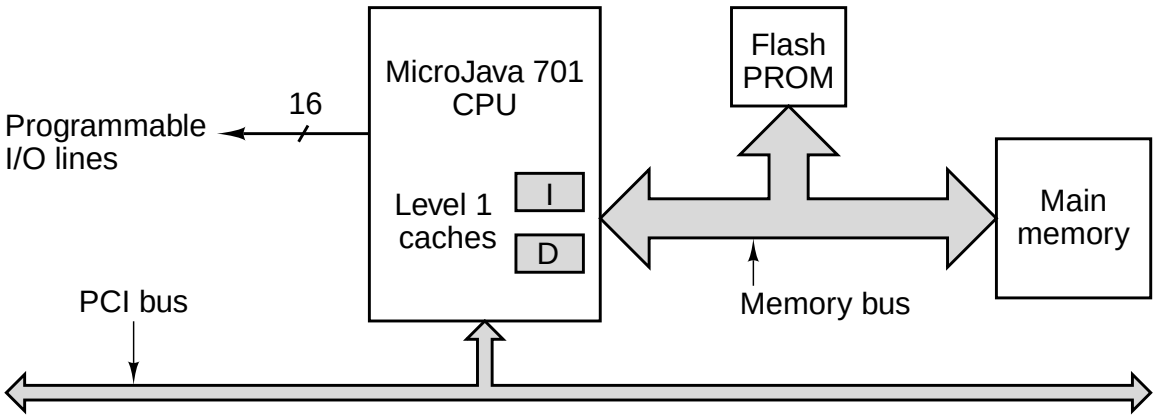


Figure 3-48. A microJava 701 system.

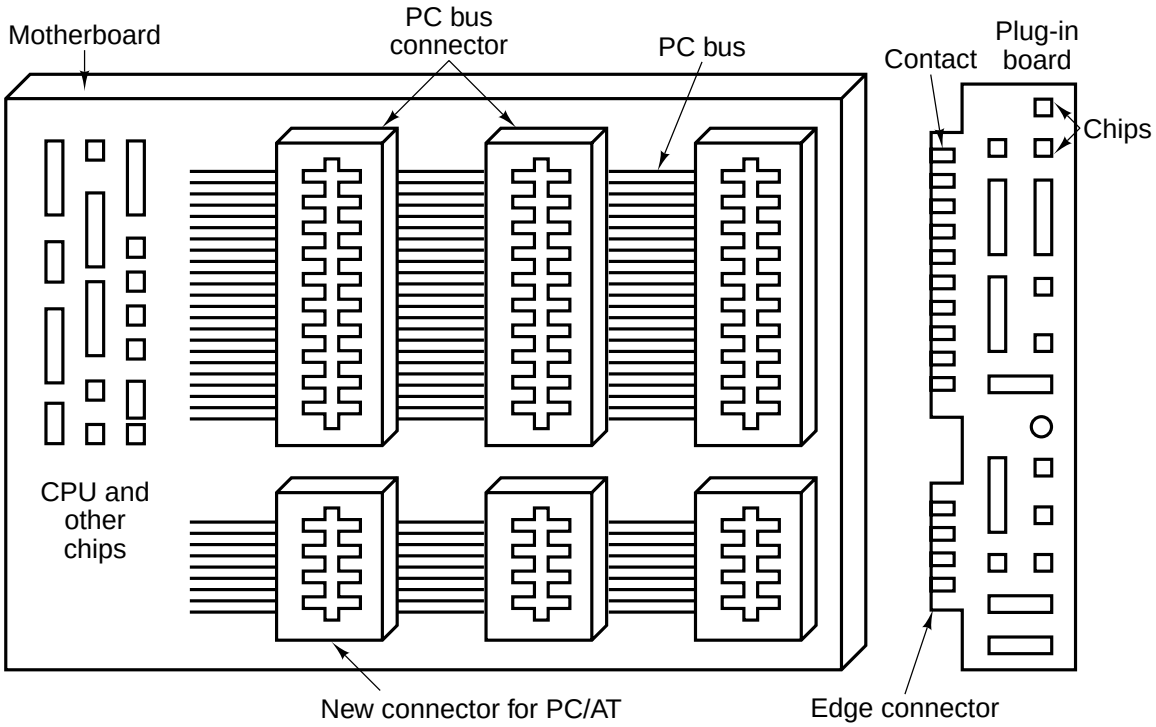


Figure 3-49. The PC/AT bus has two components, the original PC part and the new part.

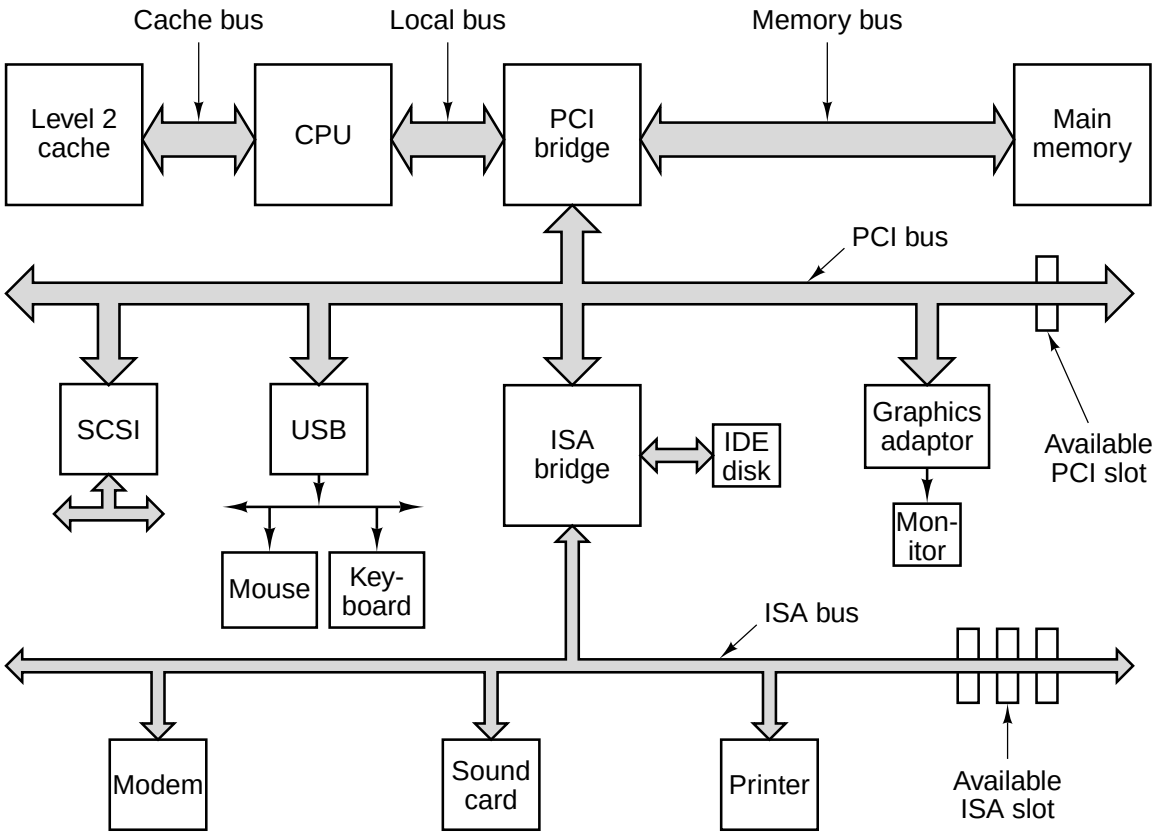


Figure 3-50. Architecture of a typical Pentium II system. The thicker buses have more bandwidth than the thinner ones.

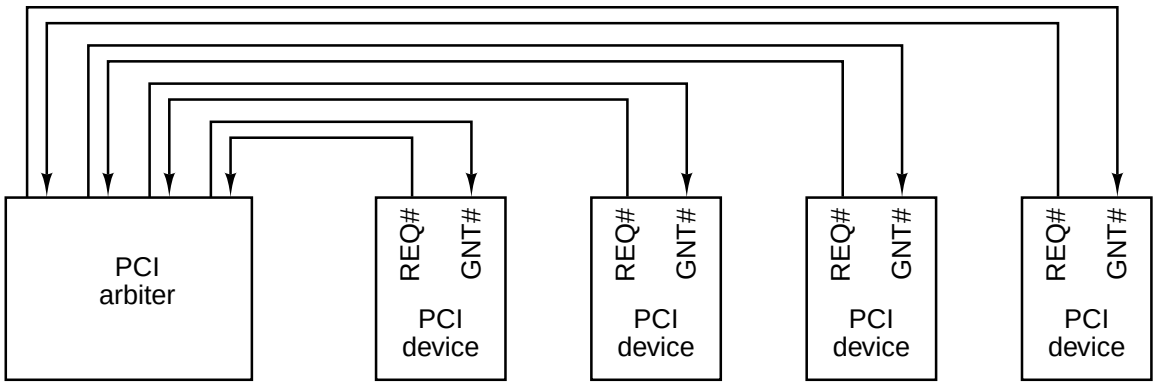


Figure 3-51. The PCI bus uses a centralized bus arbiter.

Signal	Lines	Master	Slave	Description
CLK	1			Clock (33 MHz or 66 MHz)
AD	32	×	×	Multiplexed address and data lines
PAR	1	×		Address or data parity bit
C/BE	4	×		Bus command/bit map for bytes enabled
FRAME#	1	×		Indicates that AD and C/BE are asserted
IRDY#	1	×		Read: master will accept; write: data present
IDSEL	1	×		Select configuration space instead of memory
DEVSEL#	1		×	Slave has decoded its address and is listening
TRDY#	1		×	Read: data present; write: slave will accept
STOP#	1		×	Slave wants to stop transaction immediately
PERR#	1			Data parity error detected by receiver
SERR#	1			Address parity error or system error detected
REQ#	1			Bus arbitration: request for bus ownership
GNT#	1			Bus arbitration: grant of bus ownership
RST#	1			Reset the system and all devices

(a)

Sign	Lines	Master	Slave	Description
REQ64#	1	×		Request to run a 64-bit transaction
ACK64#	1		×	Permission is granted for a 64-bit transaction
AD	32	×		Additional 32 bits of address or data
PAR64	1	×		Parity for the extra 32 address/data bits
C/BE#	4	×		Additional 4 bits for byte enables
LOCK	1	×		Lock the bus to allow multiple transactions
SBO#	1			Hit on a remote cache (for a multiprocessor)
SDONE	1			Snooping done (for a multiprocessor)
INTx	4			Request an interrupt
JTAG	5			IEEE 1149.1 JTAG test signals
M66EN	1			Wired to power or ground (66 MHz or 33 MHz)

(b)

Figure 3-52. (a) Mandatory PCI bus signals. (b) Optional PCI bus signals.

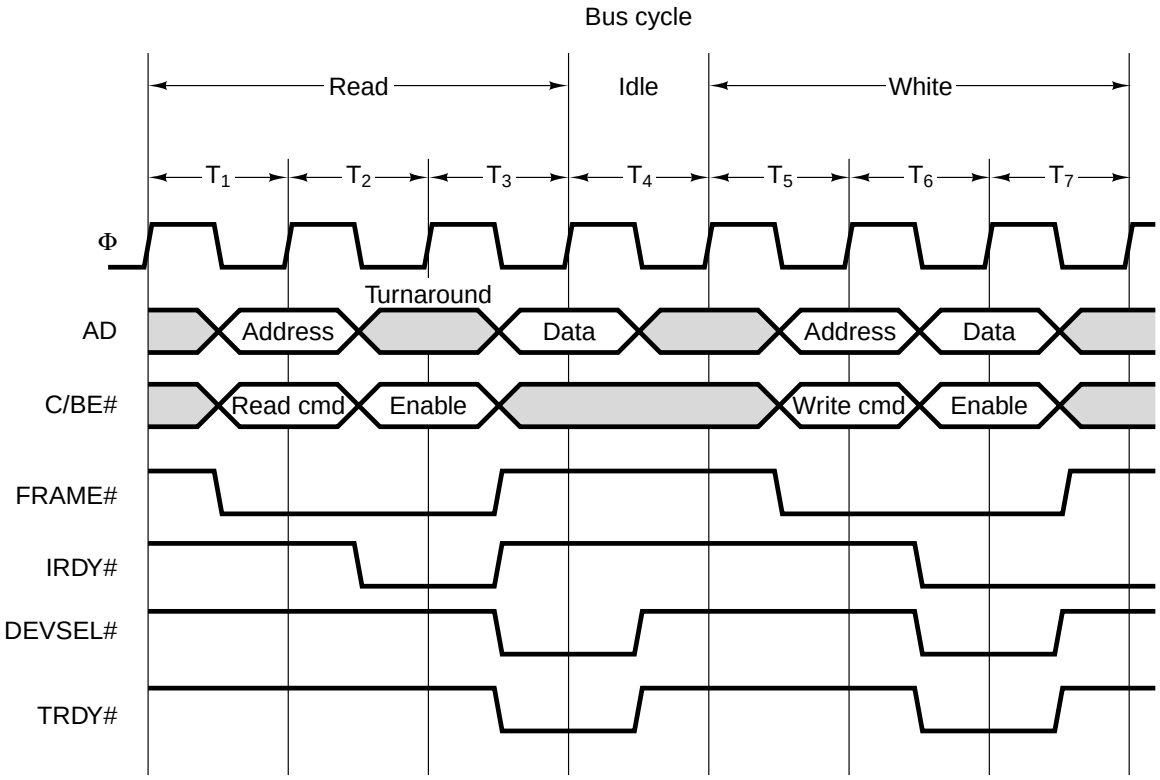


Figure 3-53. Examples of 32-bit PCI bus transactions. The first three cycles are used for a read operation, then an idle cycle, and then three cycles for a write operation.

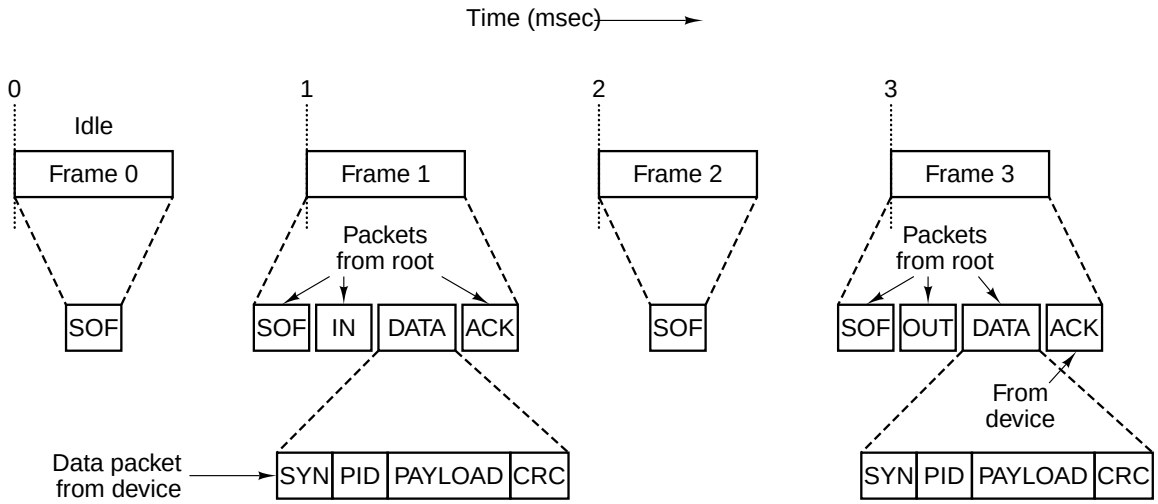


Figure 3-54. The USB root hub sends out frames every 1.00 msec.

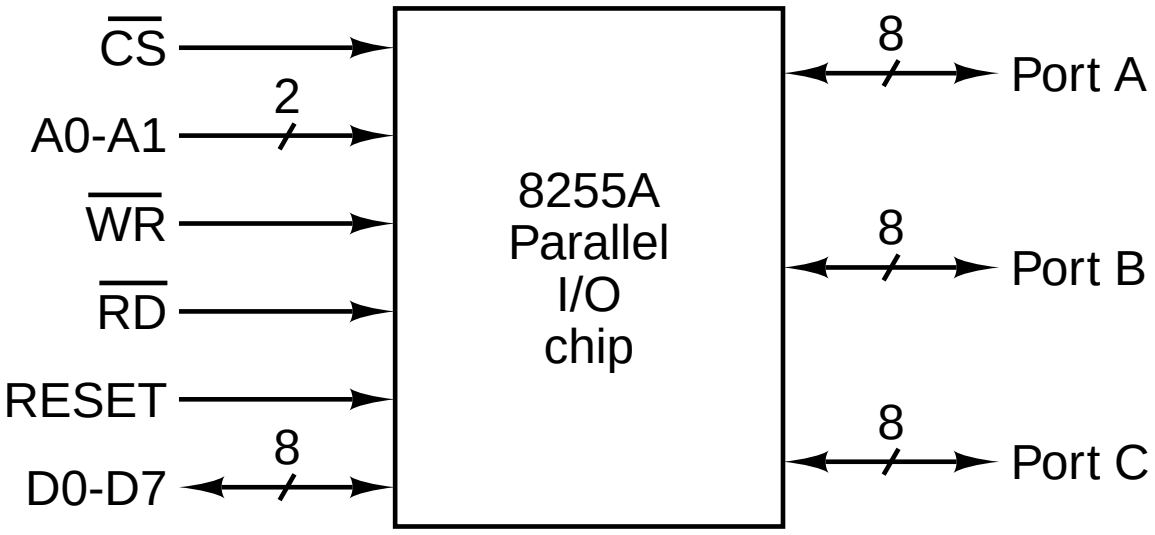


Figure 3-55. An 8255A PIO chip.

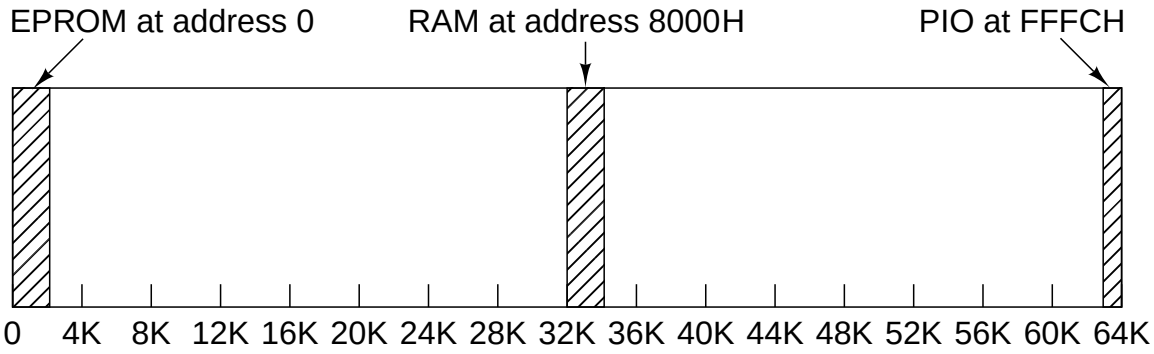


Figure 3-56. Location of the EPROM, RAM, and PIO in our 64K address space.

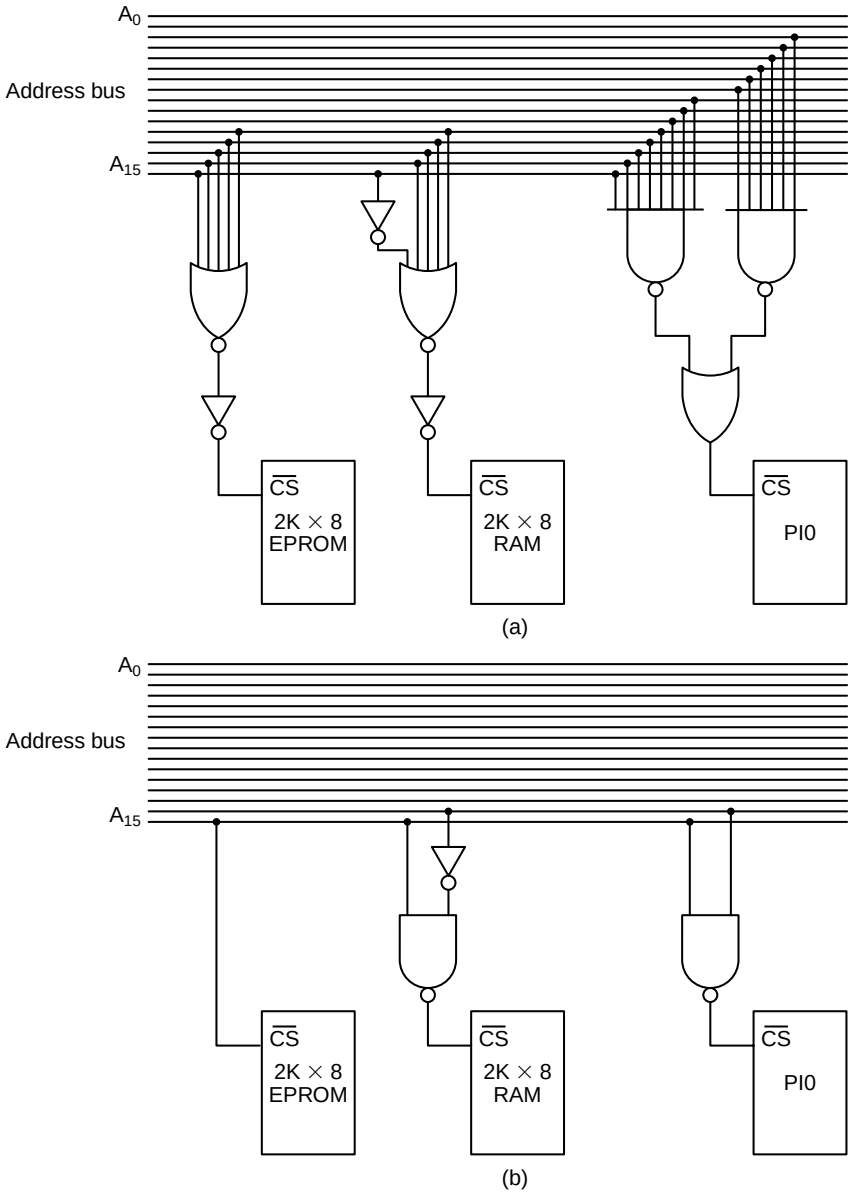


Figure 3-57. (a) Full address decoding. (b) Partial address decoding.

3.6 EXAMPLE BUSES

Buses are the glue that hold computer systems together. In this section we will take a close look at some popular buses: the ISA bus, the PCI bus, and the Universal Serial Bus. The ISA bus is a slight expansion of the original IBM PC bus. For reasons of backward compatibility, it is still present in all Intel-based PCs. However, these machines invariably have a second, faster bus as well: the PCI bus. The PCI bus is wider than the ISA bus and runs at a higher clock rate. The Universal serial bus is an increasingly popular I/O bus for low-speed peripherals such as mice and keyboards. In the following sections, we will look at each of these buses in turn.

3.6.1 The ISA Bus

The IBM PC bus was the de facto standard on 8088-based systems because nearly all PC clone vendors copied it in order to allow the many existing third-party I/O boards to be used with their systems. It had 62 signal lines, including 20 for a memory address, 8 for data, and one each for asserting memory read, memory write, I/O read, and I/O write. There were also signals for requesting and granting interrupts and using DMA, and that was about it. It was a very simple bus.

Physically, the bus was etched onto the PC's motherboard, with about half a dozen connectors spaced 2 cm apart into which cards could be inserted. Each card had a tab on it that fit in the connector. The tab had 31 gold-plated strips on each side that made electrical contact with the connector.

When IBM introduced the 80286-based PC/AT, it had a major problem on its hands. If it had started from scratch and designed an entirely new 16-bit bus, many potential customers would have hesitated to buy it because none of the vast number of PC plug-in boards available from third-party vendors would have worked using the new machine. On the other hand, sticking with the PC bus and its 20 address lines and 8 data lines would not have taken advantage of the 80286's ability to address 16M of memory and transfer 16-bit words.

The solution chosen was to extend the PC bus. PC plug-in cards have an edge connector with 62 contacts, but this edge connector does not run the full length of the board. The PC/AT solution was to put a second edge connector on the bottom of the board, adjacent to the main one, and design the AT circuitry to work with both types of boards. The general idea is illustrated in Fig. 3-1.

The second connector on the PC/AT bus contains 36 lines. Of these, 31 are provided for more address lines, more data lines, more interrupt lines, and more DMA channels, as well as power and ground. The rest deal with differences between 8-bit and 16-bit transfers.

When IBM brought out the PS/2 series as the successor to the PC and PC/AT, it decided that it was time to start again. Part of this decision may have been

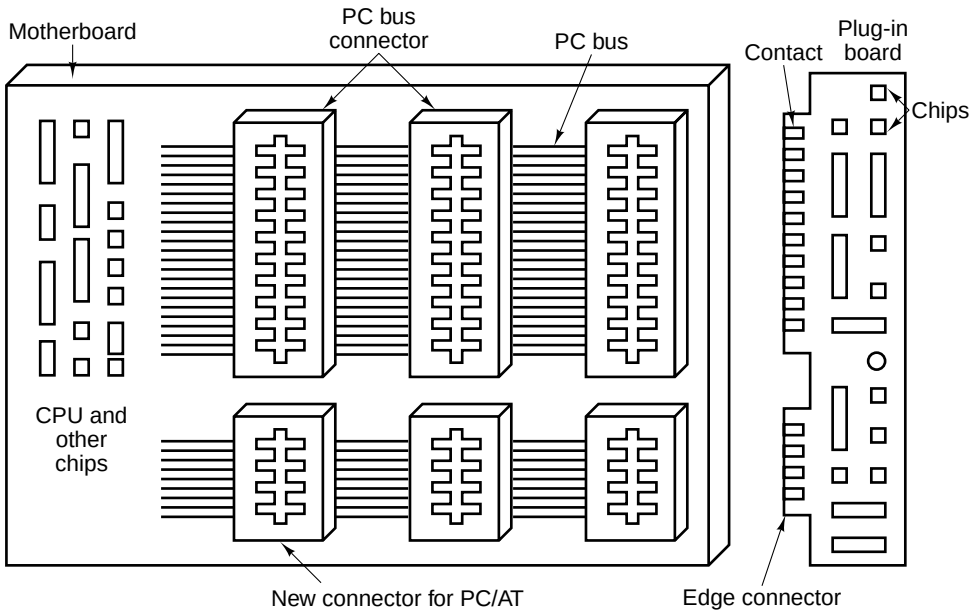


Figure 3-1. The PC/AT bus has two components, the original PC part and the new part.

technical (the PC bus was by this time really obsolete), but part was no doubt caused by a desire to put an obstacle in the way of companies making PC clones, which had taken over an uncomfortably large part of the market. Thus the mid- and upper-range PS/2 machines were equipped with a bus, the Microchannel, that was completely new, and which was protected by a wall of patents backed by an army of lawyers.

The rest of the personal computer industry reacted to this development by adopting its own standard, the **ISA** (**Industry Standard Architecture**) bus, which is basically the PC/AT bus running at 8.33 MHz. The advantage of this approach is that it retains compatibility with existing machines and cards. It also is based on a bus that IBM had liberally licensed to many companies in order to ensure that as many third parties as possible produced cards for the original PC, something that has come back to haunt IBM. Every Intel-based PC still has this bus present, although usually with one or more other buses as well. A thorough description of it can be found in (Shanley and Anderson, 1995a).

Later, the ISA bus was extended to 32 bits with a few new features thrown in (e.g., for multiprocessing). This new bus was called the **EISA** (**Extended ISA**) bus. Few boards have been produced for it, however.

3.6.2 The PCI Bus

On the original IBM PC, most applications were text based. Gradually, with the introduction of Windows, graphical user interfaces came into use. None of these applications put much of a strain on the ISA bus. However, as time went on and many applications, especially multimedia games, began to use computers to display full-screen, full-motion video, the situation changed radically.

Let us make a simple calculation. Consider a 1024×768 screen used for true color (3 bytes/pixel) moving images. One screen contains 2.25 MB of data. For smooth motion, at least 30 screens/sec are needed, for a data rate of 67.5 MB/sec. In fact, it is worse than this, since to display a video from a hard disk, CD-ROM, or DVD, the data must pass from the disk drive over the bus to the memory. Then for the display, the data must travel over the bus again to the graphics adapter. Thus we need a bus bandwidth of 135 MB/sec for the video alone, not counting the bandwidth the CPU and other devices need.

The ISA bus runs at a maximum rate of 8.33 MHz, and can transfer 2 bytes per cycle, for a maximum bandwidth of 16.7 MB/sec. The EISA bus can move 4 bytes per cycle, to achieve 33.3 MB/sec. Clearly, neither of these is even close to what is needed for full-screen video.

In 1990, Intel saw this coming and designed a new bus with a much higher bandwidth than even the EISA bus. It was called the **PCI bus** (**Peripheral Component Interconnect bus**). To encourage its use, Intel patented the PCI bus and then put all the patents into the public domain, so any company could build peripherals for it without having to pay royalties. Intel also formed an industry consortium, the PCI Special Interest Group, to manage the future of the PCI bus. As a result of these actions, the PCI bus has become extremely popular. Virtually every Intel-based computer since the Pentium has a PCI bus, and many other computers do, too. Sun even has a version of the UltraSPARC that uses the PCI bus, the UltraSPARC IIi. The PCI bus is covered in gory detail in (Shanley and Anderson, 1995b; and Solari and Willse, 1998).

The original PCI bus transferred 32 bits per cycle and ran at 33 MHz (30 nsec cycle time) for a total bandwidth of 133 MB/sec. In 1993, PCI 2.0 was introduced, and in 1995, PCI 2.1 came out. PCI 2.2 has features for mobile computers (mostly for saving battery power). The PCI bus runs at up to 66 MHz and can handle 64-bit transfers, for a total bandwidth of 528 MB/sec. With this kind of capacity, full-screen, full-motion video is doable (assuming the disk and the rest of the system are up to the job). In any event, the PCI bus will not be the bottleneck.

Even though 528 MB/sec sounds pretty fast, it still has two problems. First, it is not good enough for a memory bus. Second, it is not compatible with all those old ISA cards out there. The solution Intel thought of was to design computers with three or more buses, as shown in Fig. 3-2. Here we see that the CPU can talk to the main memory on a special memory bus, and that an ISA bus can be

connected to the PCI bus. This arrangement meets all requirements, and as a consequence, virtually all Pentium II computers use this architecture.

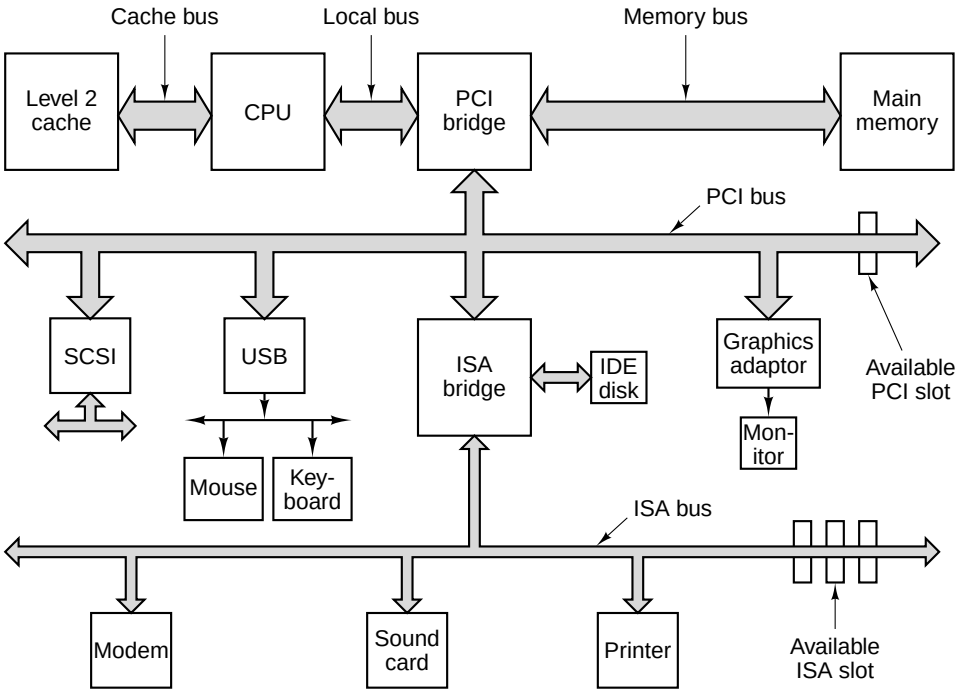


Figure 3-2. Architecture of a typical Pentium II system. The thicker buses have more bandwidth than the thinner ones.

Two key components in this architecture are the two bridge chips (which Intel manufactures—hence its interest in this whole project). The PCI bridge connects the CPU, memory and PCI bus. The ISA bridge connects the PCI bus to the ISA bus and also supports one or two IDE disks. Nearly all Pentium II systems come with one or more free PCI slots for adding new high-speed peripherals, and one or more ISA slots, for adding low-speed peripherals.

The big advantage of Fig. 3-2 is that the CPU has an extremely high bandwidth to memory, using a proprietary memory bus the PCI bus has a very high bandwidth for fast peripherals such as SCSI disks, graphics adaptors, etc., and old ISA cards can still be used. The USB box in the figure refers to the Universal Serial Bus, which will be discussed later in this chapter.

Although we have illustrated a system with one PCI bus and one ISA bus, it is possible to have multiple instances of each. There are PCI-to-PCI bridge chips available, which connect two PCI buses together, so larger systems can have two or more independent PCI buses. It is also possible to have two or more PCI-to-ISA bridge chips in a system, allowing multiple ISA buses.

It would have been nice had there been one kind of PCI card. Unfortunately, such is not the case. Options are provided for voltage, width, and timing. Older computers often use 5 volts and newer ones tend to use 3.3 volts, so the PCI bus supports both. The connectors are the same except for two bits of plastic that are there to prevent people from inserting a 5-volt card in a 3.3-volt PCI bus or vice versa. Fortunately, universal cards, which support both voltages and can plug into either kind of slot, exist. In addition to the voltage option, cards come in 32-bit and 64-bit versions. The 32-bit cards have 120 pins; the 64-bit cards have the same 120 pins plus an additional 64 pins, analogous to the way the IBM PC bus was extended to 16 bits (see Fig. 3-1). A PCI bus system that supports 64-bit cards can also take 32-bit cards, but the reverse is not true. Finally, PCI buses and cards can run at either 33 MHz or 66 MHz. The choice is made by having one pin wired either to the power supply or wired to ground. The connectors are identical for both speeds.

The PCI bus is synchronous, like all PC buses going back to the original IBM PC. All transactions on the PCI bus are between a master, officially called the **initiator**, and a slave, officially called the **target**. To keep the PCI pin count down, the address and data lines are multiplexed. In this way, only 64 pins are needed on PCI cards for address plus data signals, even though PCI supports 64-bit addresses and 64-bit data.

The multiplexed address and data pins work as follows. On a read operation, during cycle 1, the master puts the address onto the bus. On cycle 2 the master removes the address and the bus is turned around so the slave can use it. On cycle 3, the slave outputs the data requested. On write operations, the bus does not have to be turned around because the master puts on both the address and the data. Nevertheless, the minimum transaction is still three cycles. If the slave is not able to respond in three cycles, it can insert wait states. Block transfers of unlimited size are also allowed, as well as several other kinds of bus cycles.

PCI Bus Arbitration

To use the PCI bus, a device must first acquire it. PCI bus arbitration uses a centralized bus arbiter, as shown in Fig. 3-3. In most designs, the bus arbiter is built into one of the bridge chips. Every PCI device has two dedicated lines running from it to the arbiter. One line, REQ#, is used to request the bus. The other line, GNT#, is used to receive bus grants.

To request the bus, a PCI device (including the CPU), asserts REQ# and waits until it sees its GNT# line asserted by the arbiter. When that event happens, the device can use the bus on the next cycle. The algorithm used by the arbiter is not defined by the PCI specification. Round robin arbitration, priority arbitration, and other schemes are all allowed. Clearly, a good arbiter will be fair, so as not to let some devices wait forever.

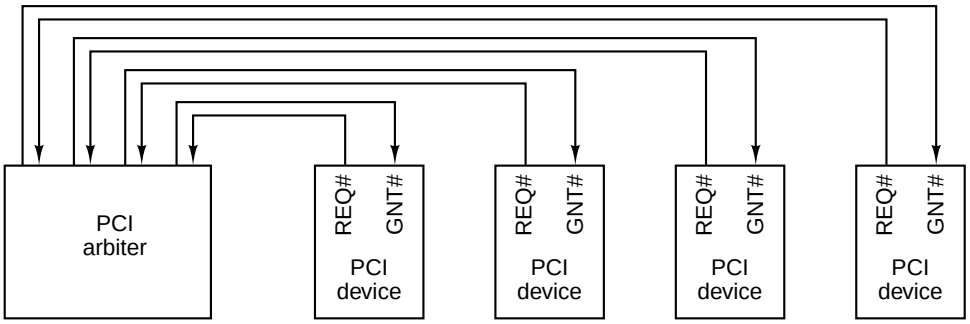


Figure 3-3. The PCI bus uses a centralized bus arbiter.

A bus grant is for one transaction, although the length of this transaction is theoretically unbounded. If a device wants to run a second transaction and no other device is requesting the bus, it can go again, although normally one idle cycle between transactions has to be inserted. However, under special circumstances, in the absence of competition for the bus, a device can make back-to-back transactions without having to insert an idle cycle. If a bus master is making a very long transfer and some other device has requested the bus, the arbiter can negate the GNT# line. The current bus master is expected to monitor the GNT# line, so when it sees the negation, it must release the bus on the next cycle. This scheme allows very long transfers (which are efficient) when there is only one candidate bus master but still gives fast response to competing devices.

PCI Bus Signals

The PCI bus has a number of mandatory signals, shown in Fig. 3-4(a), and a number of optional signals, shown in Fig. 3-4(b). The remainder of the 120 or 184 pins are used for power, ground, and related miscellaneous functions and are not listed here. The *Master* (initiator) and *Slave* (target) columns tell who asserts the signal on a normal transaction. If the signal is asserted by a different device (e.g., CLK), both columns are left blank.

Let us now look at each of the PCI bus signals briefly. We will start with the mandatory (32-bit) signals; then we will move on to the optional (64-bit) signals. The CLK signal drives the bus. Most of the other signals are synchronous with it. In contrast to the ISA bus, a PCI bus transaction begins at the falling edge of CLK, which is in the middle of the cycle, rather than at the start.

The 32 AD signals are for the address and data (for 32-bit transactions). Generally, during cycle 1 the address is asserted and during cycle 3 the data are asserted. The PAR signal is a parity bit for AD. The C/BE# signal is used for two different things. On cycle 1, it contains the bus command (read 1 word, block

Signal	Lines	Master	Slave	Description
CLK	1			Clock (33 MHz or 66 MHz)
AD	32	×	×	Multiplexed address and data lines
PAR	1	×		Address or data parity bit
C/BE	4	×		Bus command/bit map for bytes enabled
FRAME#	1	×		Indicates that AD and C/BE are asserted
IRDY#	1	×		Read: master will accept; write: data present
IDSEL	1	×		Select configuration space instead of memory
DEVSEL#	1		×	Slave has decoded its address and is listening
TRDY#	1		×	Read: data present; write: slave will accept
STOP#	1		×	Slave wants to stop transaction immediately
PERR#	1			Data parity error detected by receiver
SERR#	1			Address parity error or system error detected
REQ#	1			Bus arbitration: request for bus ownership
GNT#	1			Bus arbitration: grant of bus ownership
RST#	1			Reset the system and all devices

(a)

Sign	Lines	Master	Slave	Description
REQ64#	1	×		Request to run a 64-bit transaction
ACK64#	1		×	Permission is granted for a 64-bit transaction
AD	32	×		Additional 32 bits of address or data
PAR64	1	×		Parity for the extra 32 address/data bits
C/BE#	4	×		Additional 4 bits for byte enables
LOCK	1	×		Lock the bus to allow multiple transactions
SBO#	1			Hit on a remote cache (for a multiprocessor)
SDONE	1			Snooping done (for a multiprocessor)
INTx	4			Request an interrupt
JTAG	5			IEEE 1149.1 JTAG test signals
M66EN	1			Wired to power or ground (66 MHz or 33 MHz)

(b)

Figure 3-4. (a) Mandatory PCI bus signals. (b) Optional PCI bus signals.

read, etc.). On cycle 2 it contains a bit map of 4 bits, telling which bytes of the 32-bit word are valid. Using C/BE# it is possible to read or write any 1, 2, or 3 bytes, as well as an entire word.

The FRAME# signal is asserted by the bus master to start a bus transaction. It tells the slave that the address and bus commands are now valid. On a read,

usually IRDY# is asserted at the same time as FRAME#. It says the master is ready to accept the incoming data. On a write, IRDY# is asserted later, when the data are on the bus.

The IDSEL signal relates to the fact that every PCI device must have a 256-byte configuration space that other devices can read (by asserting IDSEL). This configuration space contains properties of the device. The Plug 'n Play feature of some operating systems uses the configuration space to find out what devices are on the bus.

Now we come to signals asserted by the slave. The first of these, DEVSEL#, announces that the slave has detected its address on the AD lines and is prepared to engage in the transaction. If DEVSEL# is not asserted within a certain time limit, the master times out and assumes the device addressed is either absent or broken.

The second slave signal is TRDY#, which the slave asserts on reads to announce that the data are on the AD lines and on writes to announce that it is prepared to accept data.

The next three signals are for error reporting. The first of these is STOP#, which the slave asserts if something disastrous happens and it wants to abort the current transaction. The next one, PERR#, is used to report a data parity error on the previous cycle. For a read, it is asserted by the master; for a write it is asserted by the slave. It is up the the receiver to take the appropriate action. Finally, SERR# is for reporting address errors and system errors.

The REQ# and GNT# signals are for doing bus arbitration. These are not asserted by the current bus master, but rather by a device that wants to become bus master. The last mandatory signal is RST#, used for resetting the system, either due to the user pushing the RESET button or some system device noticing a fatal error. Asserting this signal resets all devices and reboots the computer.

Now we come to the optional signals, most of which relate to the expansion from 32 bits to 64 bits. The REQ64# and ACK64# signals allow the master to ask permission to conduct a 64-bit transaction and allow the slave to accept, respectively. The AD, PAR64, and C/BE# signals are just extensions of the corresponding 32-bit signals.

The next three signals are not related to 32 bits versus 64 bits, but to multiprocessor systems, something that PCI boards are not required to support. The LOCK signal allows the bus to be locked for multiple transactions. The next two relate to bus snooping to maintain cache coherence.

The INTx signals are for requesting interrupts. A PCI card can have up to four separate logical devices on it, and each one can have its own interrupt request line. The JTAG signals are for the IEEE 1149.1 JTAG testing procedure. Finally, the M66EN signal is either wired high or wired low, to set the clock speed. It must not change during system operation.

PCI Bus Transactions

The PCI bus is really very simple (as buses go). To get a better feel for it, consider the timing diagram of Fig. 3-5. Here we see a read transaction, followed by an idle cycle, followed by a write transaction by the same bus master.

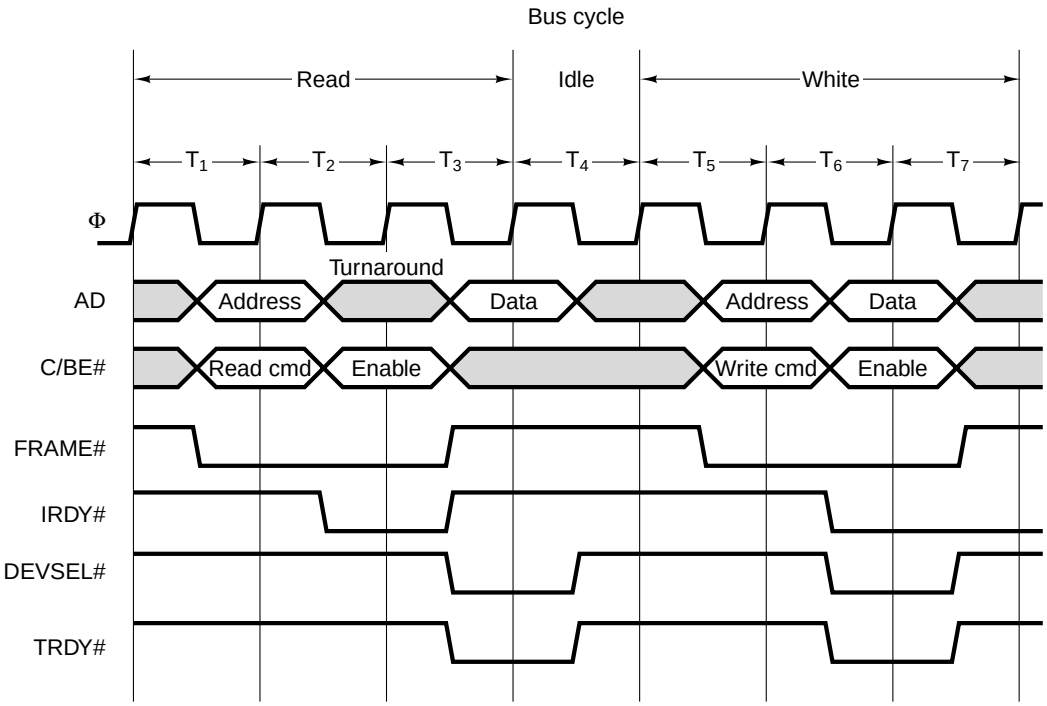


Figure 3-5. Examples of 32-bit PCI bus transactions. The first three cycles are used for a read operation, then an idle cycle, and then three cycles for a write operation.

When the falling edge of the clock happens during T_1, the master puts the memory address on AD and the bus command on C/BE#. It then asserts FRAME# to start the bus transaction.

During T_2, the master floats the address bus to let it turn around in preparation for the slave to drive it during T_3. The master also changes C/BE# to indicate which bytes in the word addressed it wants to enable (i.e., read in).

In T_3, the slave asserts DEVSEL# so the master knows it got the address and is planning to respond. It also puts the data on the AD lines and asserts TRDY# to tell the master that it has done so. If the slave was not able to respond so quickly, it would still assert DEVSEL# to announce its presence but keep TRDY# negated until it could get the data out there. This procedure would introduce one or more wait

states.

In this example (and often in reality), the next cycle is idle. Starting in T_5 we see the same master initiating a write. It starts out by putting the address and command onto the bus, as usual. Only now, in the second cycle it asserts the data. Since the same device is driving the AD lines, there is no need for a turnaround cycle. In T_7, the memory accepts the data.

3.6.3 The Universal Serial Bus

The PCI bus is fine for attaching high-speed peripherals to a computer, but it is far too expensive to have a PCI interface for each low-speed I/O device such as a keyboard or mouse. Historically, each standard I/O device was connected to the computer in a special way, with some free ISA and PCI slots for adding new devices. Unfortunately, this scheme has been fraught with problems from the beginning. For example, each new I/O device often comes with its own ISA or PCI card. The user is often responsible for setting switches and jumpers on the card and making sure the settings do not conflict with other cards. Then the user must open up the case, carefully insert the card, close the case, and reboot the computer. For many users, this process is difficult and error prone. In addition, the number of ISA and PCI slots is very limited (two or three typically). Plug 'n Play cards eliminate the jumper settings, but the user still has to open the computer to insert the card and bus slots are still limited.

To deal with this problem, in the mid 1990s, representatives from seven companies (Compaq, DEC, IBM, Intel, Microsoft, NEC, and Northern Telecom) got together to design a better way to attach low-speed I/O devices to a computer. Since then, hundreds of other companies have joined them. The resulting standard is called **USB** (**Universal Serial Bus**) and it is being widely implemented in personal computers. It is described further in (Anderson, 1997; and Tan, 1997).

Some of the goals of the companies that originally conceived of the USB and started the project were as follows:

1. Users must not have to set switches or jumpers on boards or devices.
2. Users must not have to open the case to install new I/O devices.
3. There should be only one kind of cable, good for connecting all devices.
4. I/O devices should get their power from the cable.
5. Up to 127 devices should be attachable to a single computer.
6. The system should support real-time devices (e.g., sound, telephone).
7. Devices should be installable while the computer is running.
8. No reboot should be needed after installing a new device.
9. The new bus and its I/O devices should be inexpensive to manufacture.

USB meets all these goals. It is designed for low-speed devices such as keyboards, mice, still cameras, snapshot scanners, digital telephones, and so on. The total USB bandwidth is 1.5 MB/sec, which is enough for a substantial number of these devices. This low limit was chosen to keep the cost down.

A USB system consists of a **root hub** that plugs into the main bus (see Fig. 3-2). This hub has sockets for cables that can connect to I/O devices or to expansion hubs, to provide more sockets, so the topology of a USB system is a tree with its root at the root hub, inside the computer. The cables have different connectors on the hub end and on the device end, to prevent people from accidentally connecting two hub sockets together.

The cable consists of four wires: two for data, one for power (+5 volts), and one for ground. The signaling system transmits a 0 as a voltage transition and a 1 as the absence of a voltage transition, so long runs of 0s generate a regular pulse stream.

When a new I/O device is plugged in, the root hub detects this event and interrupts the operating system. The operating system then queries the device to find out what it is and how much USB bandwidth it needs. If the operating system decides that there is enough bandwidth for the device, it assigns the new device a unique address (1 – 127) and downloads this address and other information to configuration registers inside the device. In this way, new devices can be added on-the-fly, without any user configuration required and without having to install new ISA or PCI cards. Uninitialized cards start out with address 0, so they can be addressed. To make the cabling simpler, many USB devices contain built-in hubs to accept additional USB devices. For example, a monitor might have two hub sockets to accommodate the left and right speakers.

Logically, the USB system can be viewed as a set of bit pipes from the root hub to the I/O devices. Each device can split its bit pipe up into at most 16 subpipes for different types of data (e.g., audio and video). Within each pipe or subpipe, data flows from the root hub to the device or the other way. There is no traffic between two I/O devices.

Precisely every 1.00 ± 0.05 msec, the root hub broadcasts a new frame to keep all the devices synchronized in time. A frame is associated with a bit pipe, and consists of packets, the first of which is from the root hub to the device. Subsequent packets in the frame may also be in this direction, or they may be back from the device to the root hub. A sequence of four frames is shown in Fig. 3-6. In Fig. 3-6 there is no work to be done in frames 0 and 2, so all that is needed is one SOF (Start of Frame) packet. This packet is always broadcast to all devices. Frame 1 is a poll, for example, a request to a scanner to return the bits it has found on the image it is scanning. Frame 3 consists of delivering data to some device, for example to a printer.

USB supports four kinds of frames: control, isochronous, bulk, and interrupt. Control frames are used to configure devices, give them commands, and inquire about their status. Isochronous frames are for real-time devices such as

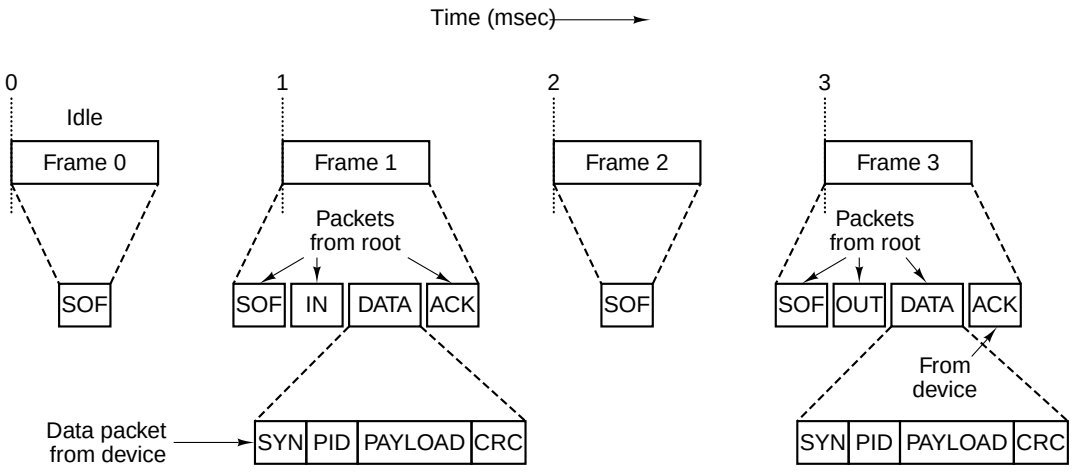


Figure 3-6. The USB root hub sounds out frames every 1.00 msec.

microphones, loudspeakers, and telephones that need to send or accept data at precise time intervals. They have a highly-predictable delay but provide no retransmissions in the event of errors. Bulk frames are for large transfers to or from devices with no real-time requirements such as printers. Finally, interrupt frames are needed because USB does not support interrupts. For example, instead of having the keyboard cause an interrupt whenever a key is struck, the operating system can poll it every 50 msec to collect any pending keystrokes.

A frame consists of one or more packets, possibly some in each direction. Four kinds of packets exist: token, data, handshake, and special. Token packets are from the root to a device and are for system control. The SOF, IN, and OUT packets in Fig. 3-6 are token packets. The SOF (Start of Frame) packet is the first one in each frame and marks the beginning of the frame. If there is no work to do, the SOF packet is the only one in the frame. The IN token packet is a poll, asking the device to return certain data. Fields in the IN packet tell which bit pipe is being polled so the device knows which data to return (if it has multiple streams). The OUT token packet announces that data for the device will follow. A fourth type of token packet, SETUP (not shown in the figure), is used for configuration.

Besides the token packet, three other kinds exist. These are DATA (used to transmit up to 64 bytes of information either way), handshake, and special packets. The format of a data packet is shown in Fig. 3-6. It consists of an 8-bit synchronization field, an 8-bit packet type (PID), the payload, and a 16-bit **CRC** (**Cyclic Redundancy Code**) to detect errors. Three kinds of handshake packets are defined: ACK (the previous data packet was correctly received), NAK (a CRC error was detected), and STALL (please wait—I am busy right now).

Now let us look at Fig. 3-6 again. Every 1.00 msec a frame must be sent from the root hub, even if there is no work. Frames 0 and 2 consist of just an SOF

packet, indicating that there was no work. Frame 1 is a poll, so it starts out with SOF and IN packets from the computer to the I/O device, followed by a DATA packet from the device to the computer. The ACK packet tells the device that the data were received correctly. In case of an error, a NAK would be sent back to the device and the packet would be retransmitted for bulk data (but not for isochronous data). Frame 3 is similar in structure to frame 1, except that now the flow of data are from the computer to the device.

4

THE MICROARCHITECTURE LEVEL

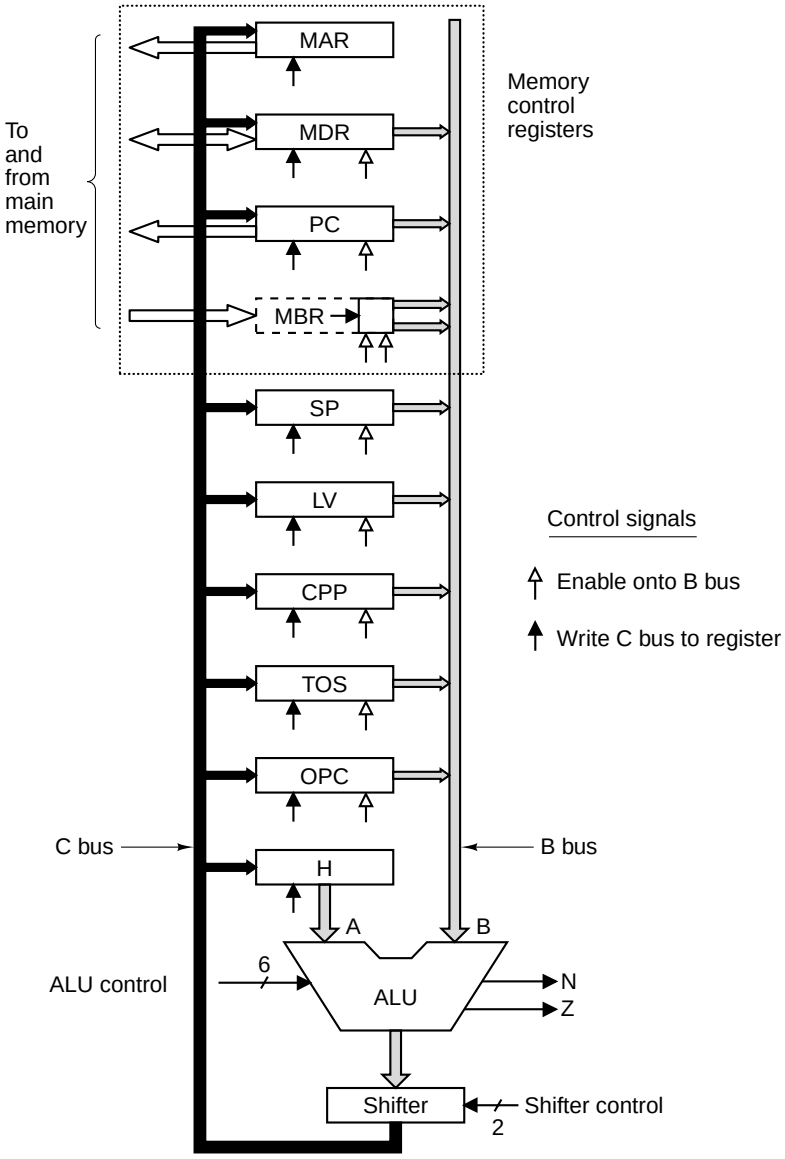


Figure 4-1. The data path of the example microarchitecture used in this chapter.

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	1	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	–1

Figure 4-2. Useful combinations of ALU signals and the function performed.

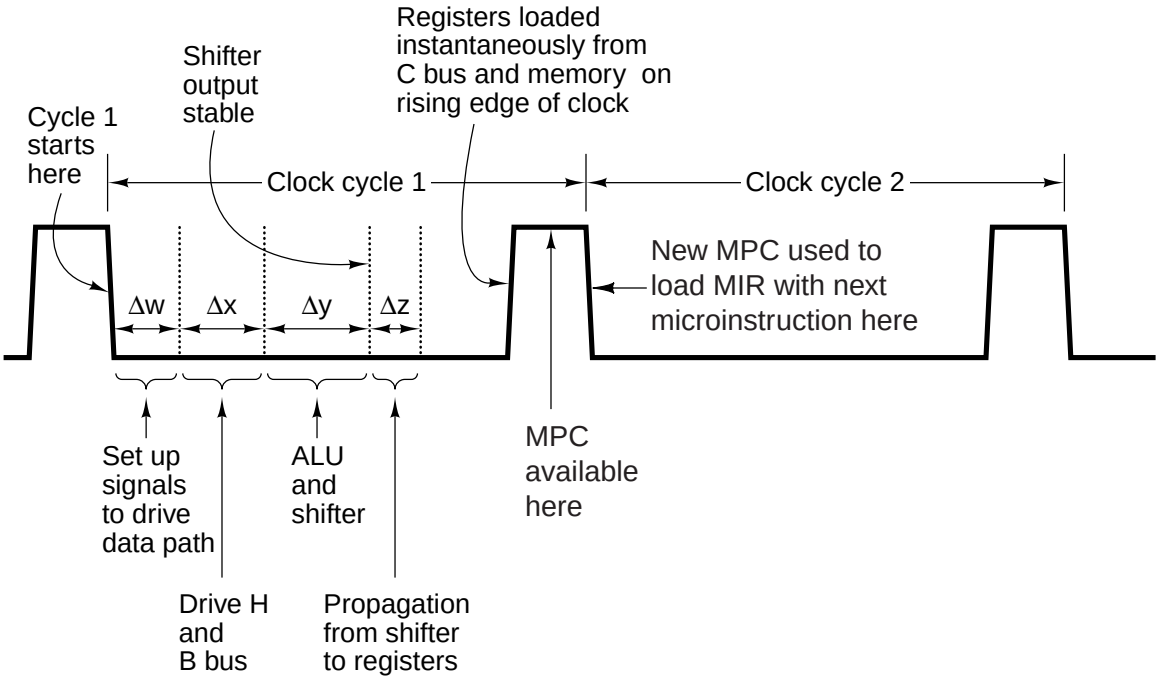


Figure 4-3. Timing diagram of one data path cycle.

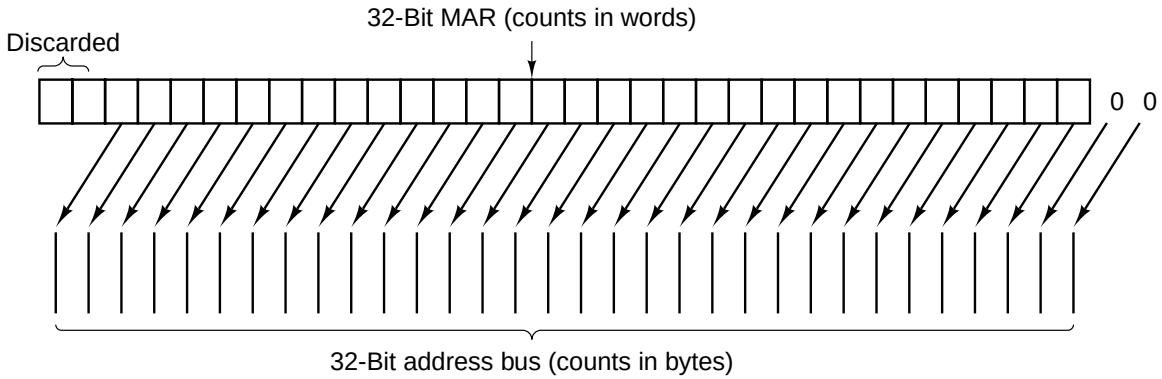


Figure 4-4. Mapping of the bits in MAR to the address bus.

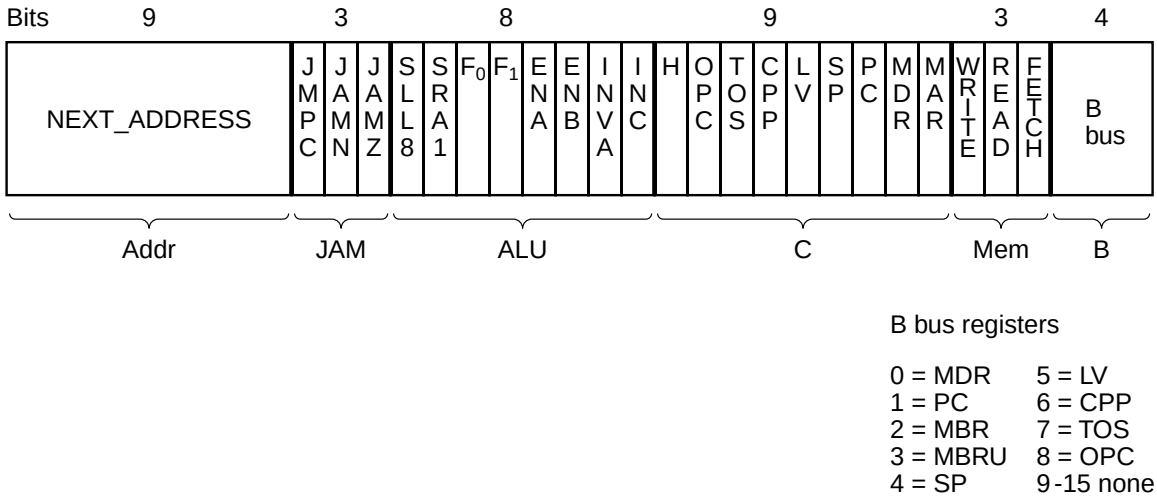


Figure 4-5. The microinstruction format for the Mic-1.

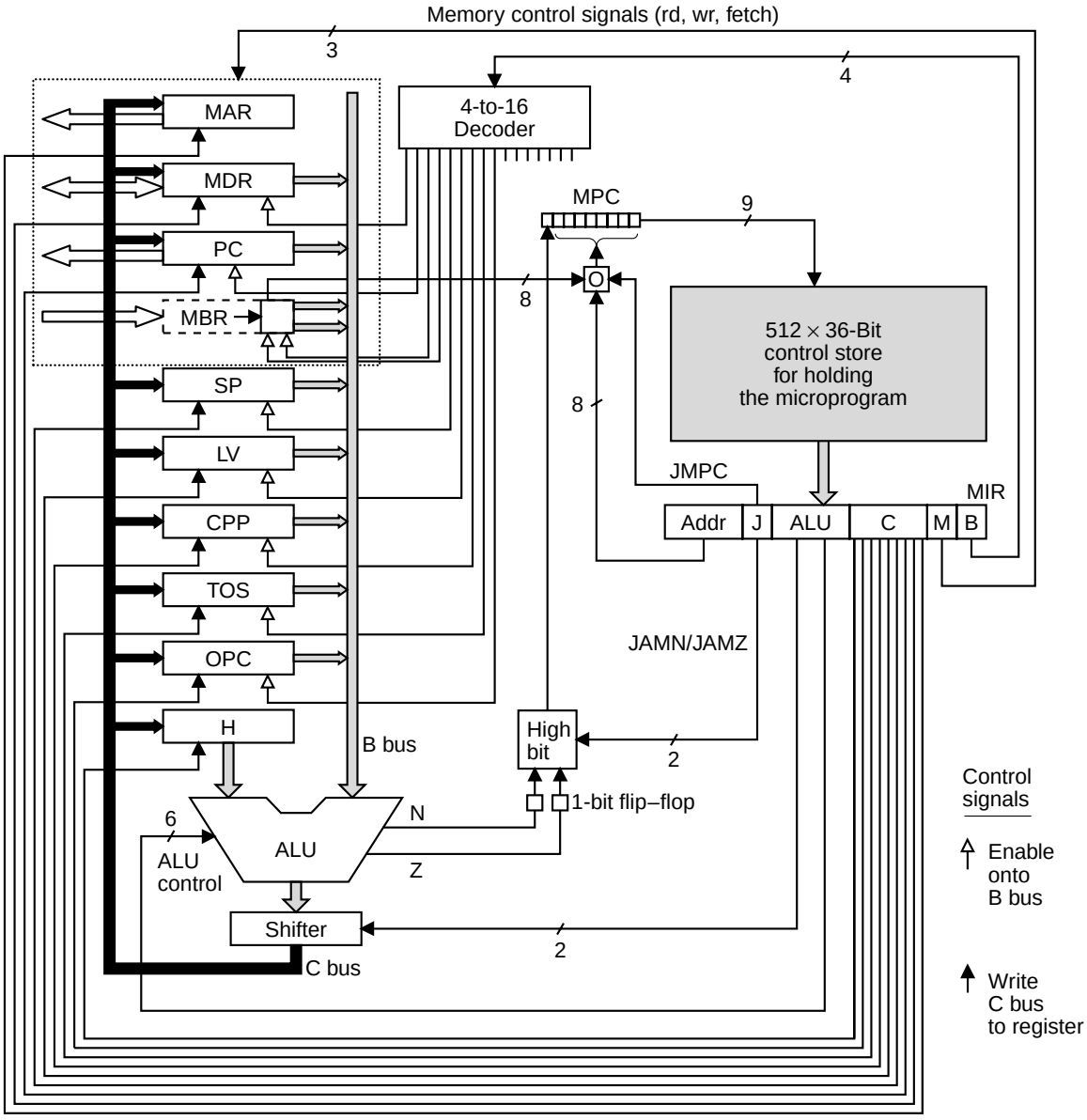


Figure 4-6. The complete block diagram of our example microarchitecture, the Mic-1.

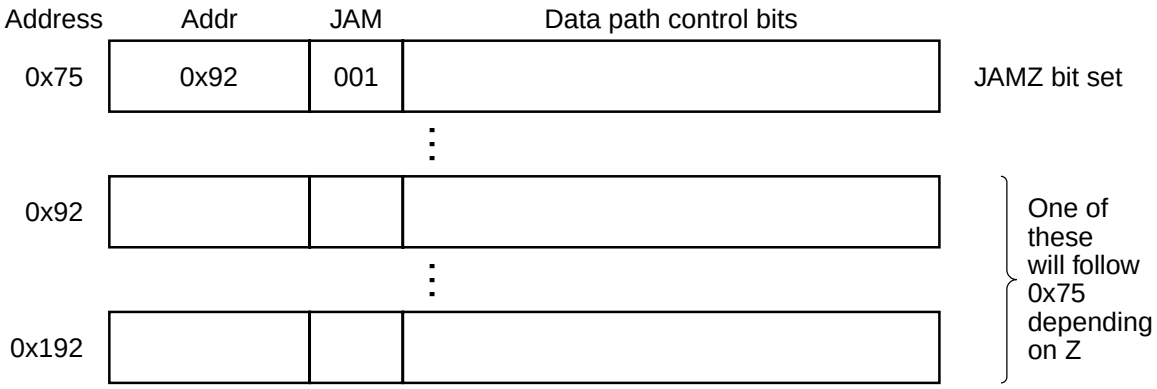


Figure 4-7. A microinstruction with JAMZ set to 1 has two potential successors.

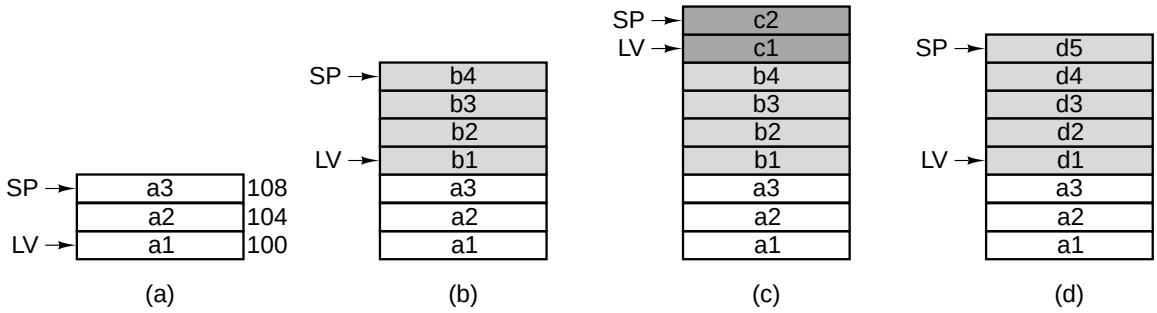


Figure 4-8. Use of a stack for storing local variables. (a) While *A* is active. (b) After *A* calls *B*. (c) After *B* calls *C*. (d) After *C* and *B* return and *A* calls *D*.

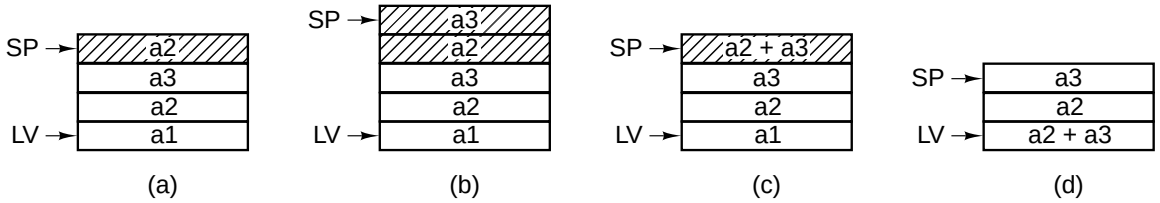


Figure 4-9. Use of an operand stack for doing an arithmetic computation.

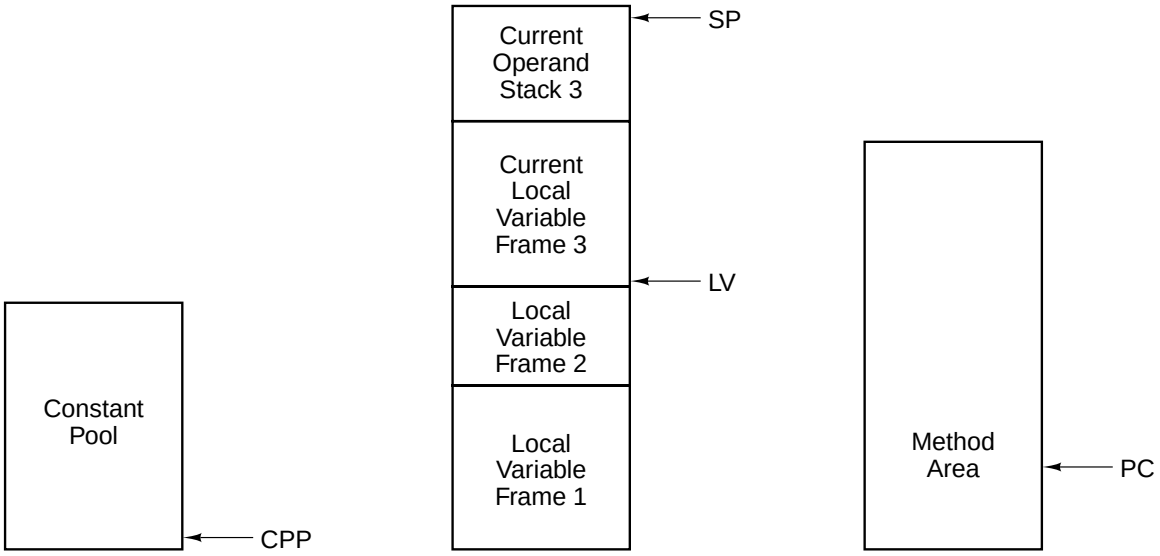


Figure 4-10. The various parts of the IJVM memory.

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Figure 4-11. The IJVM instruction set. The operands *byte*, *const*, and *varnum* are 1 byte. The operands *disp*, *index*, and *offset* are 2 bytes.

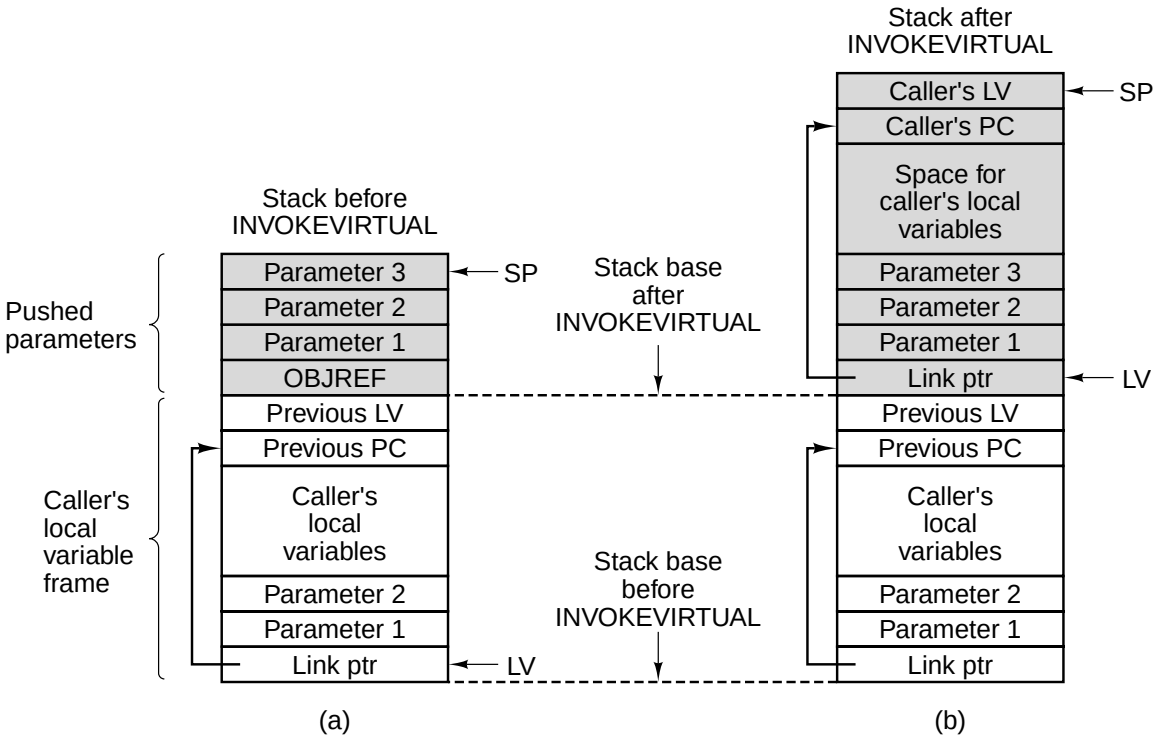


Figure 4-12. (a) Memory before executing INVOKEVIRTUAL. (b) After executing it.

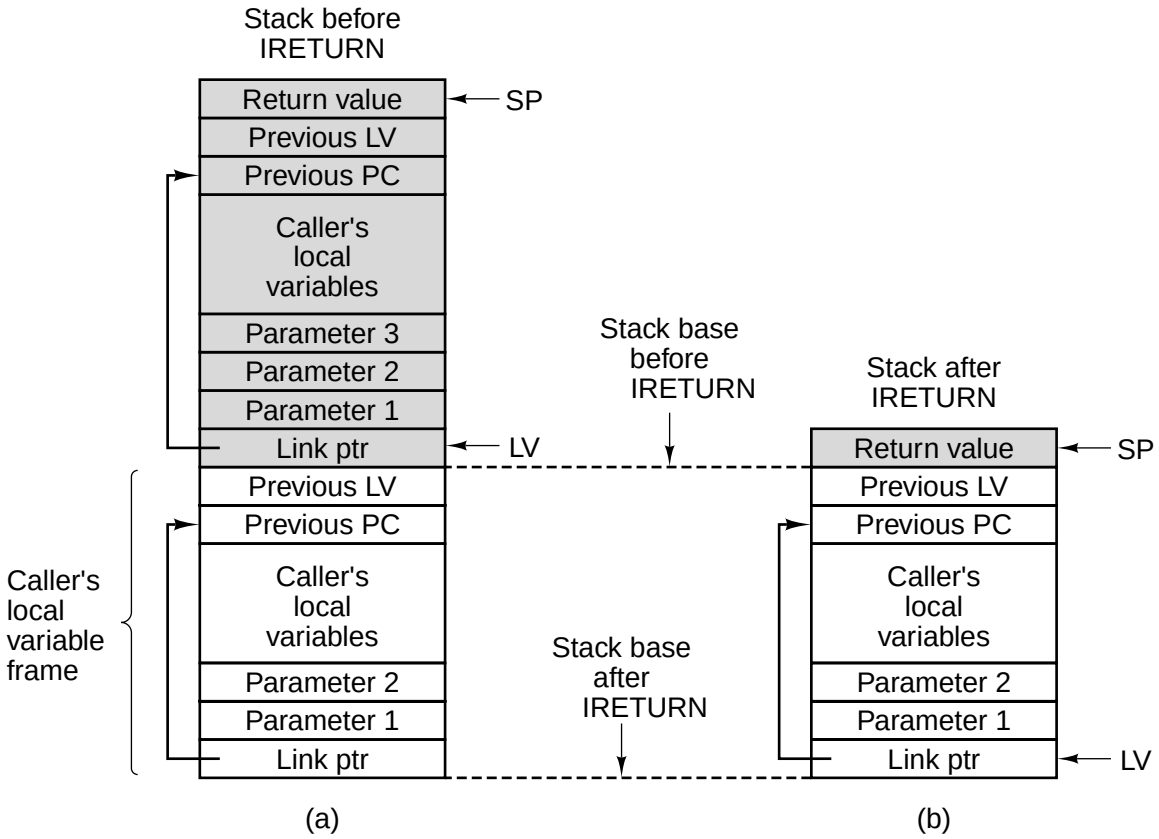


Figure 4-13. (a) Memory before executing IRETURN. (b) After executing it.

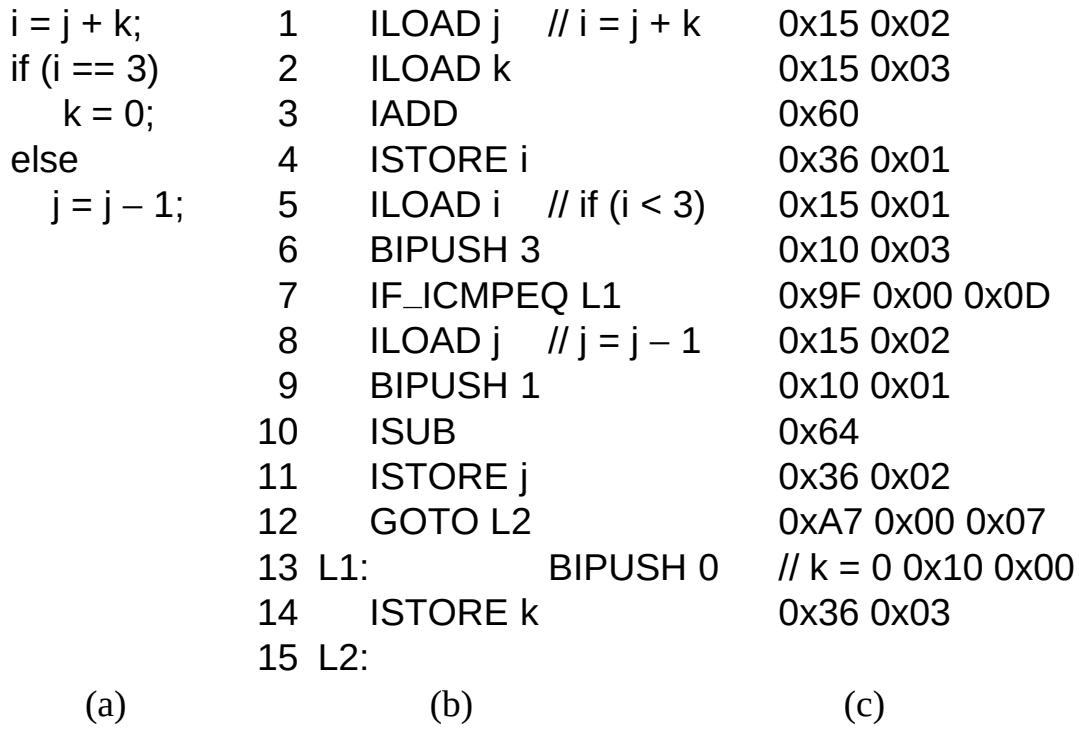

i = j + k;
if (i == 3)
k = 0;
else
j = j – 1;
1 ILOAD j // i = j + k 0x15 0x02
2 ILOAD k 0x15 0x03
3 IADD 0x60
4 ISTORE i 0x36 0x01
5 ILOAD i // if (i < 3) 0x15 0x01
6 BIPUSH 3 0x10 0x03
7 IF_ICMPEQ L1 0x9F 0x00 0x0D
8 ILOAD j // j = j – 1 0x15 0x02
9 BIPUSH 1 0x10 0x01
10 ISUB 0x64
11 ISTORE j 0x36 0x02
12 GOTO L2 0xA7 0x00 0x07
13 L1: BIPUSH 0 // k = 0 0x10 0x00
14 ISTORE k 0x36 0x03
15 L2:
(a)
(b)
(c)

Figure 4-14. (a) A Java fragment. (b) The corresponding Java assembly language. (c) The IJVM program in hexadecimal.

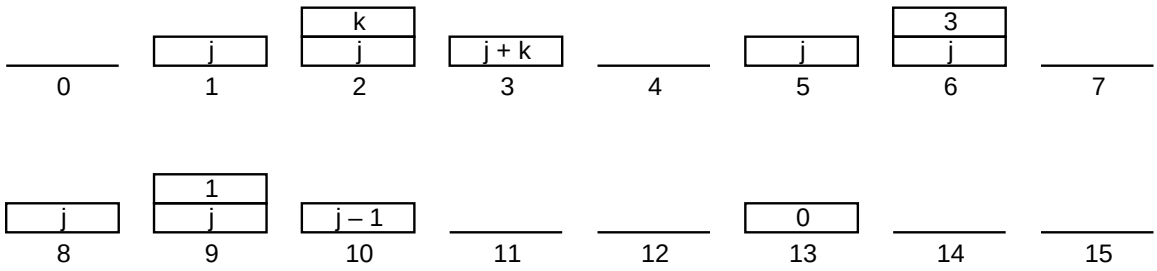


Figure 4-15. The stack after each instruction of Fig. 4-14(b).

DEST = H
DEST = SOURCE
DEST = $\overline{H}$
DEST = $\overline{SOURCE}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE – 1
DEST = –H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = –1

Figure 4-16. All permitted operations. Any of the above operations may be extended by adding ‘‘<< 8’’ to them to shift the result left by 1 byte. For example, a common operation is H = MBR < < 8

Label	Operations	Comments
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch
nop1	goto Main1	Do nothing
iadd1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Add top two words; write to top of stack
isub1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
isub2	H = TOS	H = top of stack
isub3	MDR = TOS = MDR – H; wr; goto Main1	Do subtraction; write to top of stack
iand1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iand2	H = TOS	H = top of stack
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Do AND; write to new top of stack
ior1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
ior2	H = TOS	H = top of stack
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Do OR; write to new top of stack
dup1	MAR = SP = SP + 1	Increment SP and copy to MAR
dup2	MDR = TOS; wr; goto Main1	Write new stack word
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
pop2		Wait for new TOS to be read from memory
pop3	TOS = MDR; goto Main1	Copy new word to TOS
swap1	MAR = SP – 1; rd	Set MAR to SP – 1; read 2nd word from stack
swap2	MAR = SP	Set MAR to top word
swap3	H = MDR; wr	Save TOS in H; write 2nd word to top of stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP – 1; wr	Set MAR to SP – 1; write as 2nd word on stack
swap6	TOS = H; goto Main1	Update TOS
bipush1	SP = MAR = SP + 1	MBR = the byte to push onto stack
bipush2	PC = PC + 1; fetch	Increment PC, fetch next opcode
bipush3	MDR = TOS = MBR; wr; goto Main1	Sign-extend constant and push on stack
iload1	H = LV	MBR contains index; copy LV to H
iload2	MAR = MBRU + H; rd	MAR = address of local variable to push
iload3	MAR = SP = SP + 1	SP points to new top of stack; prepare write
iload4	PC = PC + 1; fetch; wr	Inc PC; get next opcode; write top of stack
iload5	TOS = MDR; goto Main1	Update TOS
istore1	H = LV	MBR contains index; Copy LV to H
istore2	MAR = MBRU + H	MAR = address of local variable to store into
istore3	MDR = TOS; wr	Copy TOS to MDR; write word
istore4	SP = MAR = SP – 1; rd	Read in next-to-top word on stack
istore5	PC = PC + 1; fetch	Increment PC; fetch next opcode
istore6	TOS = MDR; goto Main1	Update TOS

wide1	PC = PC + 1; fetch; goto (MBR OR 0x100)	Multiway branch with high bit set
wide_iload1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_iload2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_iload3	H = MBRU OR H	H = 16-bit index of local variable
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = address of local variable to push
wide_istore1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_istore2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_istore3	H = MBRU OR H	H = 16-bit index of local variable
wide_istore4	MAR = LV + H; goto istore3	MAR = address of local variable to store into
ldc_w1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
ldc_w2	H = MBRU << 8	H = 1st index byte << 8
ldc_w3	H = MBRU OR H	H = 16-bit index into constant pool
ldc_w4	MAR = H + CPP; rd; goto iload3	MAR = address of constant in pool

Figure 4-17. The microprogram for the Mic-1 (part 1 of 3).

Label	Operations	Comments
iinc1	H = LV	MBR contains index; Copy LV to H
iinc2	MAR = MBRU + H; rd	Copy LV + index to MAR; Read variable
iinc3	PC = PC + 1; fetch	Fetch constant
iinc4	H = MDR	Copy variable to H
iinc5	PC = PC + 1; fetch	Fetch next opcode
iinc6	MDR = MBR + H; wr; goto Main1	Put sum in MDR; update variable
goto1	OPC = PC – 1	Save address of opcode.
goto2	PC = PC + 1; fetch	MBR = 1st byte of offset; fetch 2nd byte
goto3	H = MBR << 8	Shift and save signed first byte in H
goto4	H = MBRU OR H	H = 16-bit branch offset
goto5	PC = OPC + H; fetch	Add offset to OPC
goto6	goto Main1	Wait for fetch of next opcode
iflt1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iflt2	OPC = TOS	Save TOS in OPC temporarily
iflt3	TOS = MDR	Put new top of stack in TOS
iflt4	N = OPC; if (N) goto T; else goto F	Branch on N bit
ifeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
ifeq2	OPC = TOS	Save TOS in OPC temporarily
ifeq3	TOS = MDR	Put new top of stack in TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Branch on Z bit
if_icmpeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
if_icmpeq2	MAR = SP = SP – 1	Set MAR to read in new top-of-stack
if_icmpeq3	H = MDR; rd	Copy second stack word to H
if_icmpeq4	OPC = TOS	Save TOS in OPC temporarily
if_icmpeq5	TOS = MDR	Put new top of stack in TOS
if_icmpeq6	Z = OPC – H; if (Z) goto T; else goto F	If top 2 words are equal, goto T, else goto F
T	OPC = PC – 1; fetch; goto goto2	Same as goto1; needed for target address
F	PC = PC + 1	Skip first offset byte
F2	PC = PC + 1; fetch	PC now points to next opcode
F3	goto Main1	Wait for fetch of opcode

invokevirtual1	PC = PC + 1; fetch	MBR = index byte 1; inc. PC, get 2nd byte
invokevirtual2	H = MBRU << 8	Shift and save first byte in H
invokevirtual3	H = MBRU OR H	H = offset of method pointer from CPP
invokevirtual4	MAR = CPP + H; rd	Get pointer to method from CPP area
invokevirtual5	OPC = PC + 1	Save Return PC in OPC temporarily
invokevirtual6	PC = MDR; fetch	PC points to new method; get param count
invokevirtual7	PC = PC + 1; fetch	Fetch 2nd byte of parameter count
invokevirtual8	H = MBRU << 8	Shift and save first byte in H
invokevirtual9	H = MBRU OR H	H = number of parameters
invokevirtual10	PC = PC + 1; fetch	Fetch first byte of # locals
invokevirtual11	TOS = SP – H	TOS = address of OBJREF – 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = address of OBJREF (new LV)
invokevirtual13	PC = PC + 1; fetch	Fetch second byte of # locals
invokevirtual14	H = MBRU << 8	Shift and save first byte in H
invokevirtual15	H = MBRU OR H	H = # locals
invokevirtual16	MDR = SP + H + 1; wr	Overwrite OBJREF with link pointer
invokevirtual17	MAR = SP = MDR;	Set SP, MAR to location to hold old PC
invokevirtual18	MDR = OPC; wr	Save old PC above the local variables
invokevirtual19	MAR = SP = SP + 1	SP points to location to hold old LV
invokevirtual20	MDR = LV; wr	Save old LV above saved PC
invokevirtual21	PC = PC + 1; fetch	Fetch first opcode of new method.
invokevirtual22	LV = TOS; goto Main1	Set LV to point to LV Frame

Figure 4-17. The microprogram for the Mic-1 (part 2 of 3).

Label	Operations	Comments
ireturn1	MAR = SP = LV; rd	Reset SP, MAR to get link pointer
ireturn2		Wait for read
ireturn3	LV = MAR = MDR; rd	Set LV to link ptr; get old PC
ireturn4	MAR = LV + 1	Set MAR to read old LV
ireturn5	PC = MDR; rd; fetch	Restore PC; fetch next opcode
ireturn6	MAR = SP	Set MAR to write TOS
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto Main1	Save return value on original top of stack

Figure 4-17. The microprogram for the Mic-1 (part 3 of 3).

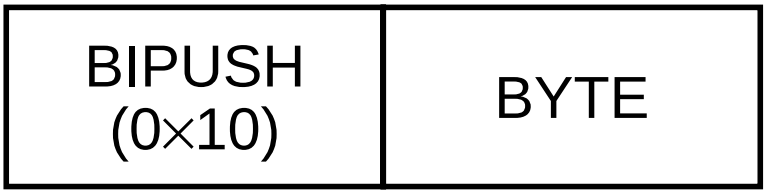


Figure 4-18. The BIPUSH instruction format.

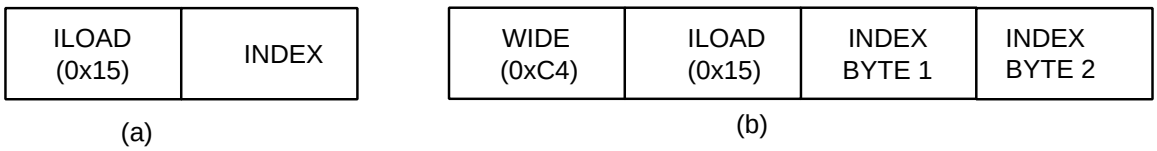


Figure 4-19. (a) ILOAD with a 1-byte index. (b) WIDE ILOAD with a 2-byte index.

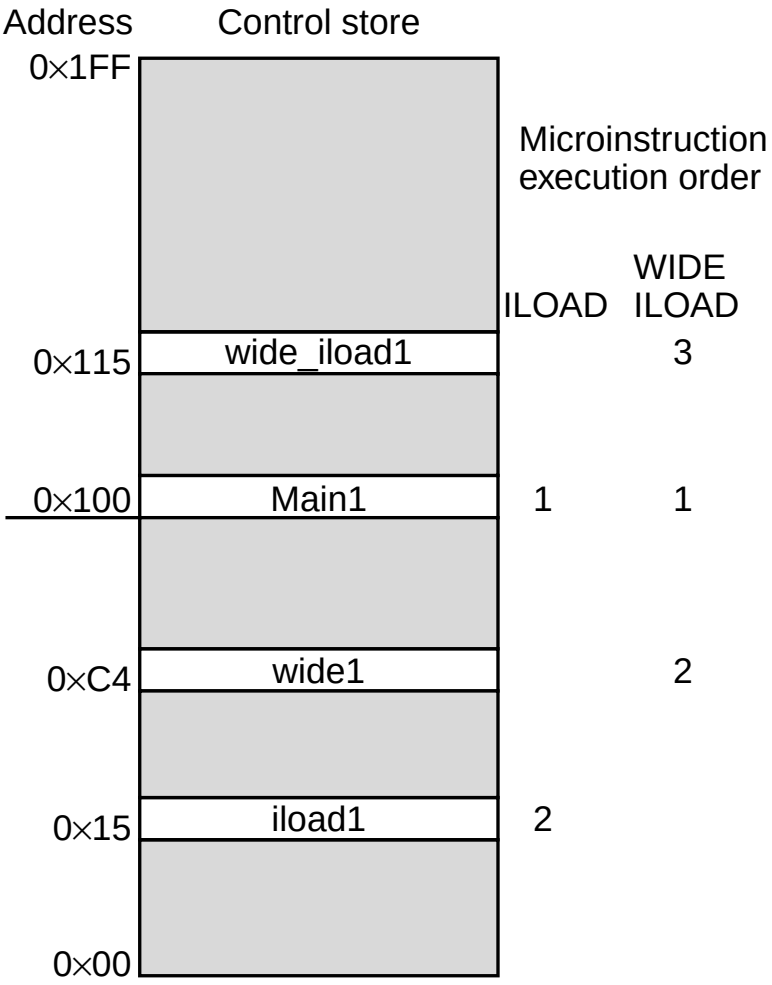


Figure 4-20. The initial microinstruction sequence for ILOAD and WIDE ILOAD. The addresses are examples.

Figure 4-21. The IINC instruction has two different operand fields.

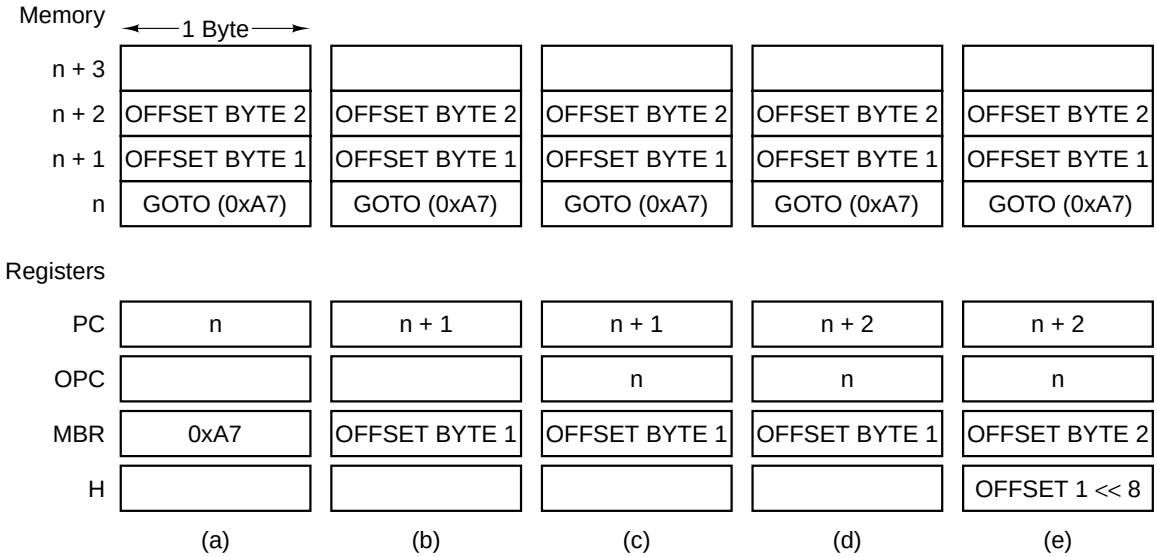


Figure 4-22. The situation at the start of various microinstructions. (a) Main1. (b) goto1. (c) goto2. (d) goto3. (e) goto4.

Label	Operations	Comments
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
pop2		Wait for new TOS to be read from memory
pop3	TOS = MDR; goto Main1	Copy new word to TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch

Figure 4-23. New microprogram sequence for executing POP.

Label	Operations	Comments
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
Main1.pop	PC = PC + 1; fetch	MBR holds opcode; fetch next byte
pop3	TOS = MDR; goto (MBR)	Copy new word to TOS; dispatch on opcode

Figure 4-24. Enhanced microprogram sequence for executing POP.

Label	Operations	Comments
iload1	H = LV	MBR contains index; Copy LV to H
iload2	MAR = MBRU + H; rd	MAR = address of local variable to push
iload3	MAR = SP = SP + 1	SP points to new top of stack; prepare write
iload4	PC = PC + 1; fetch; wr	Inc PC; get next opcode; write top of stack
iload5	TOS = MDR; goto Main1	Update TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch

Figure 4-25. Mic-1 code for executing ILOAD.

Label	Operations	Comments
iload1	MAR = MBRU + LV; rd	MAR = address of local variable to push
iload2	MAR = SP = SP + 1	SP points to new top of stack; prepare write
iload3	PC = PC + 1; fetch; wr	Inc PC; get next opcode; write top of stack
iload4	TOS = MDR	Update TOS
iload5	PC = PC + 1; fetch; goto (MBR)	MBR already holds opcode; fetch index byte

Figure 4-26. Three-bus code for executing ILOAD.

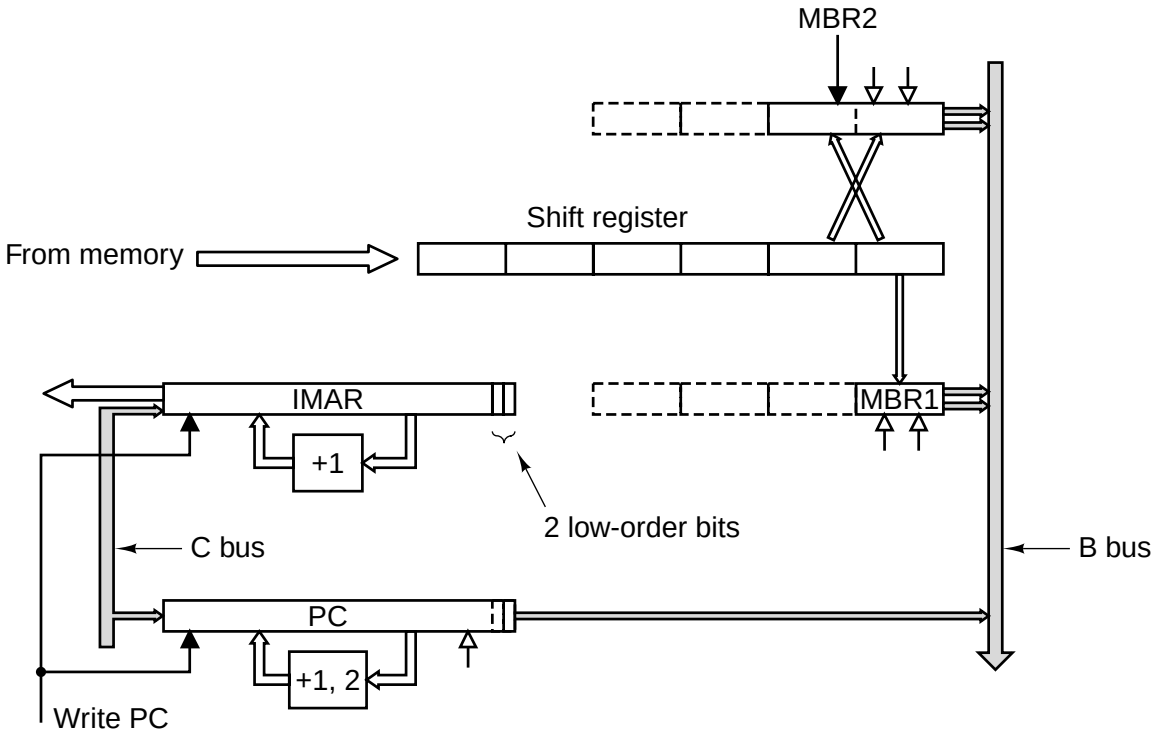
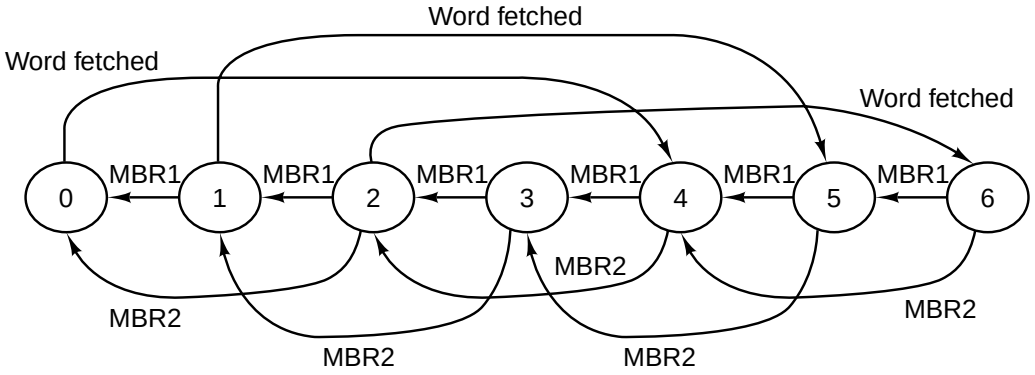


Figure 4-27. A fetch unit for the Mic-1.



Transitions

MBR1: Occurs when MBR1 is read
MBR2: Occurs when MBR2 is read
Word fetched: Occurs when a memory word is read and 4 bytes are put into the shift register

Figure 4-28. A finite state machine for implementing the IFU.

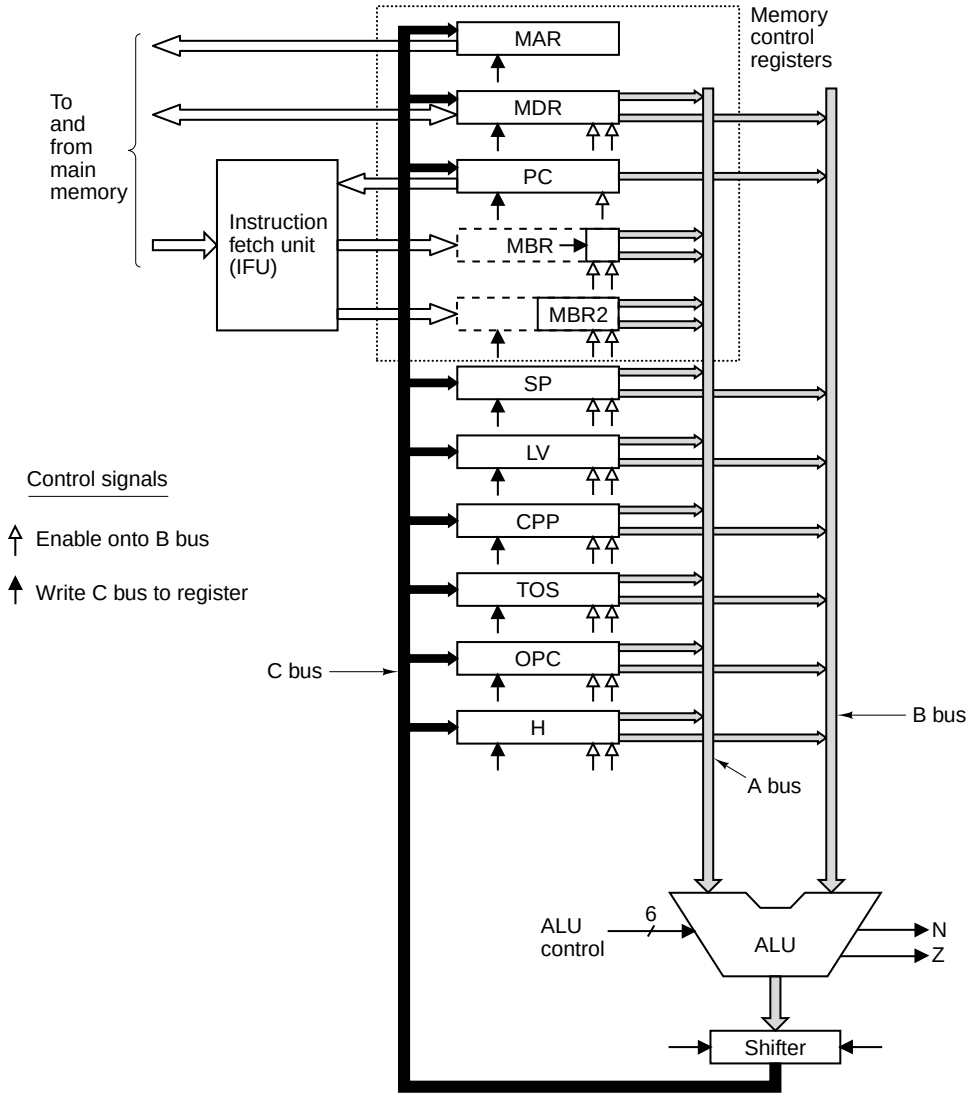


Figure 4-29. The datapath for Mic-2.

Label	Operations	Comments
nop1	goto (MBR)	Branch to next instruction
iadd1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack
iadd3	MDR = TOS = MDR+H; wr; goto (MBR1)	Add top two words; write to new top of stack
isub1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
isub2	H = TOS	H = top of stack
isub3	MDR = TOS = MDR–H; wr; goto (MBR1)	Subtract TOS from Fetched TOS-1
iand1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iand2	H = TOS	H = top of stack
iand3	MDR = TOS = MDR AND H; wr; goto (MBR1)	AND Fetched TOS-1 with TOS
ior1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
ior2	H = TOS	H = top of stack
ior3	MDR = TOS = MDR OR H; wr; goto (MBR1)	OR Fetched TOS-1 with TOS
dup1	MAR = SP = SP + 1	Increment SP; copy to MAR
dup2	MDR = TOS; wr; goto (MBR1)	Write new stack word
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
pop2		Wait for read
pop3	TOS = MDR; goto (MBR1)	Copy new word to TOS
swap1	MAR = SP – 1; rd	Read 2nd word from stack; set MAR to SP
swap2	MAR = SP	Prepare to write new 2nd word
swap3	H = MDR; wr	Save new TOS; write 2nd word to stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP – 1; wr	Write old TOS to 2nd place on stack
swap6	TOS = H; goto (MBR1)	Update TOS
bipush1	SP = MAR = SP + 1	Set up MAR for writing to new top of stack
bipush2	MDR = TOS = MBR1; wr; goto (MBR1)	Update stack in TOS and memory
iload1	MAR = LV + MBR1U; rd	Move LV + index to MAR; read operand
iload2	MAR = SP = SP + 1	Increment SP; Move new SP to MAR
iload3	TOS = MDR; wr; goto (MBR1)	Update stack in TOS and memory
istore1	MAR = LV + MBR1U	Set MAR to LV + index
istore2	MDR = TOS; wr	Copy TOS for storing
istore3	MAR = SP = SP – 1; rd	Decrement SP; read new TOS
istore4		Wait for read
istore5	TOS = MDR; goto (MBR1)	Update TOS
wide1	goto (MBR1 OR 0x100)	Next address is 0x100 Ored with opcode
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	Identical to iload1 but using 2-byte index
wide_istore1	MAR = LV + MBR2U; goto istore2	Identical to istore1 but using 2-byte index
ldc_w1	MAR = CPP + MBR2U; rd; goto iload2	Same as wide_iload1 but indexing off CPP

iinc1	MAR = LV + MBR1U; rd	Set MAR to LV + index for read
iinc2	H = MBR1	Set H to constant
iinc3	MDR = MDR + H; wr; goto (MBR1)	Increment by constant and update
goto1	H = PC – 1	Copy PC to H
goto2	PC = H + MBR2	Add offset and update PC
goto3		Have to wait for IFU to fetch new opcode
goto4	goto (MBR1)	Dispatch to next instruction
iflt1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iflt2	OPC = TOS	Save TOS in OPC temporarily
iflt3	TOS = MDR	Put new top of stack in TOS
iflt4	N = OPC; if (N) goto T; else goto F	Branch on N bit

Figure 4-30. The microprogram for the Mic-2 (part 1 of 2).

Label	Operations	Comments
ifeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
ifeq2	OPC = TOS	Save TOS in OPC temporarily
ifeq3	TOS = MDR	Put new top of stack in TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Branch on Z bit
if_icmpeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
if_icmpeq2	MAR = SP = SP – 1	Set MAR to read in new top-of-stack
if_icmpeq3	H = MDR; rd	Copy second stack word to H
if_icmpeq4	OPC = TOS	Save TOS in OPC temporarily
if_icmpeq5	TOS = MDR	Put new top of stack in TOS
if_icmpeq6	Z = H – OPC; if (Z) goto T; else goto F	If top 2 words are equal, goto T, else goto F
T	H = PC – 1; goto goto2	Same as goto1
F	H = MBR2	Touch bytes in MBR2 to discard
F2	goto (MBR1)	
invokevirtual1	MAR = CPP + MBR2U; rd	Put address of method pointer in MAR
invokevirtual2	OPC = PC	Save Return PC in OPC
invokevirtual3	PC = MDR	Set PC to 1st byte of method code.
invokevirtual4	TOS = SP – MBR2U	TOS = address of OBJREF – 1
invokevirtual5	TOS = MAR = H = TOS + 1	TOS = address of OBJREF
invokevirtual6	MDR = SP + MBR2U + 1; wr	Overwrite OBJREF with link pointer
invokevirtual7	MAR = SP = MDR	Set SP, MAR to location to hold old PC
invokevirtual8	MDR = OPC; wr	Prepare to save old PC
invokevirtual9	MAR = SP = SP + 1	Inc. SP to point to location to hold old LV
invokevirtual10	MDR = LV; wr	Save old LV
invokevirtual11	LV = TOS; goto (MBR1)	Set LV to point to zeroth parameter.
ireturn1	MAR = SP = LV; rd	Reset SP, MAR to read Link ptr
ireturn2		Wait for link ptr
ireturn3	LV = MAR = MDR; rd	Set LV, MAR to link ptr; read old PC
ireturn4	MAR = LV + 1	Set MAR to point to old LV; read old LV
ireturn5	PC = MDR; rd	Restore PC
ireturn6	MAR = SP	
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto (MBR1)	Save return value on original top of stack

Figure 4-30. The microprogram for the Mic-2 (part 2 of 2).

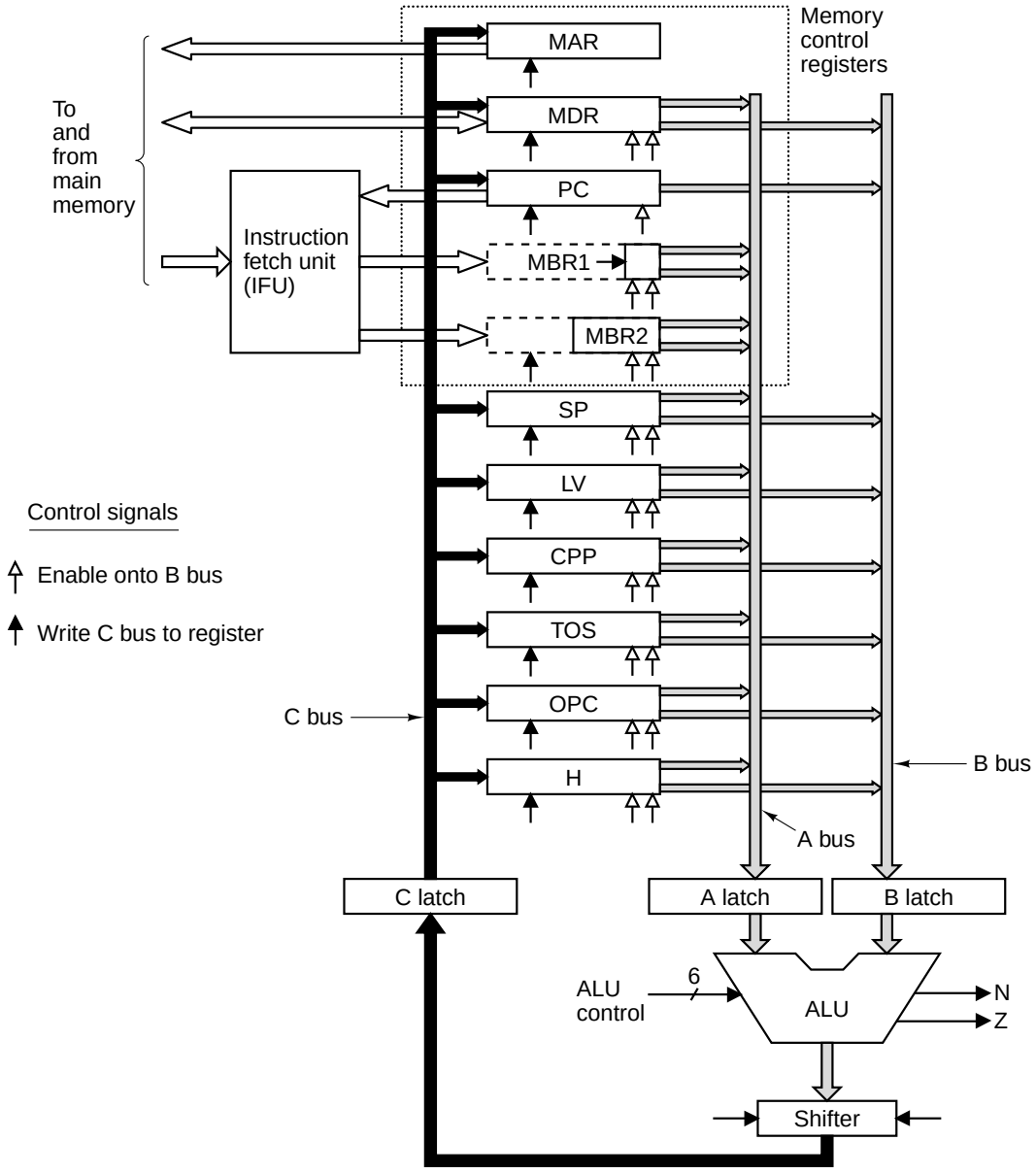


Figure 4-31. The three-bus data path used in the Mic-3.

Label	Operations	Comments
swap1	MAR = SP – 1; rd	Read 2nd word from stack; set MAR to SP
swap2	MAR = SP	Prepare to write new 2nd word
swap3	H = MDR; wr	Save new TOS; write 2nd word to stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP – 1; wr	Write old TOS to 2nd place on stack
swap6	TOS = H; goto (MBR1)	Update TOS

Figure 4-32. The Mic-2 code for SWAP.

	Swap1	Swap2	Swap3	Swap4	Swap5	Swap6	
Cy	**MAR=SP–1;rd**	**MAR=SP**	**H=MDR;wr**	**MDR=TOS**	**MAR=SP–1;wr**	**TOS=H;goto (MBR1)**	
1	B=SP						
2	C=B–1	B=SP					
3	MAR=C; rd	C=B					
4	MDR=mem	MAR=C					
5			B=MDR				
6			C=B	B=TOS			
7			H=C; wr	C=B	B=SP		
8			Mem=MDR	MDR=C	C=B–1	B=H	
9					MAR=C; wr	C=B	
10					Mem=MDR	TOS=C	
11						goto (MBR1)	

Figure 4-33. The implementation of SWAP on the Mic-3.

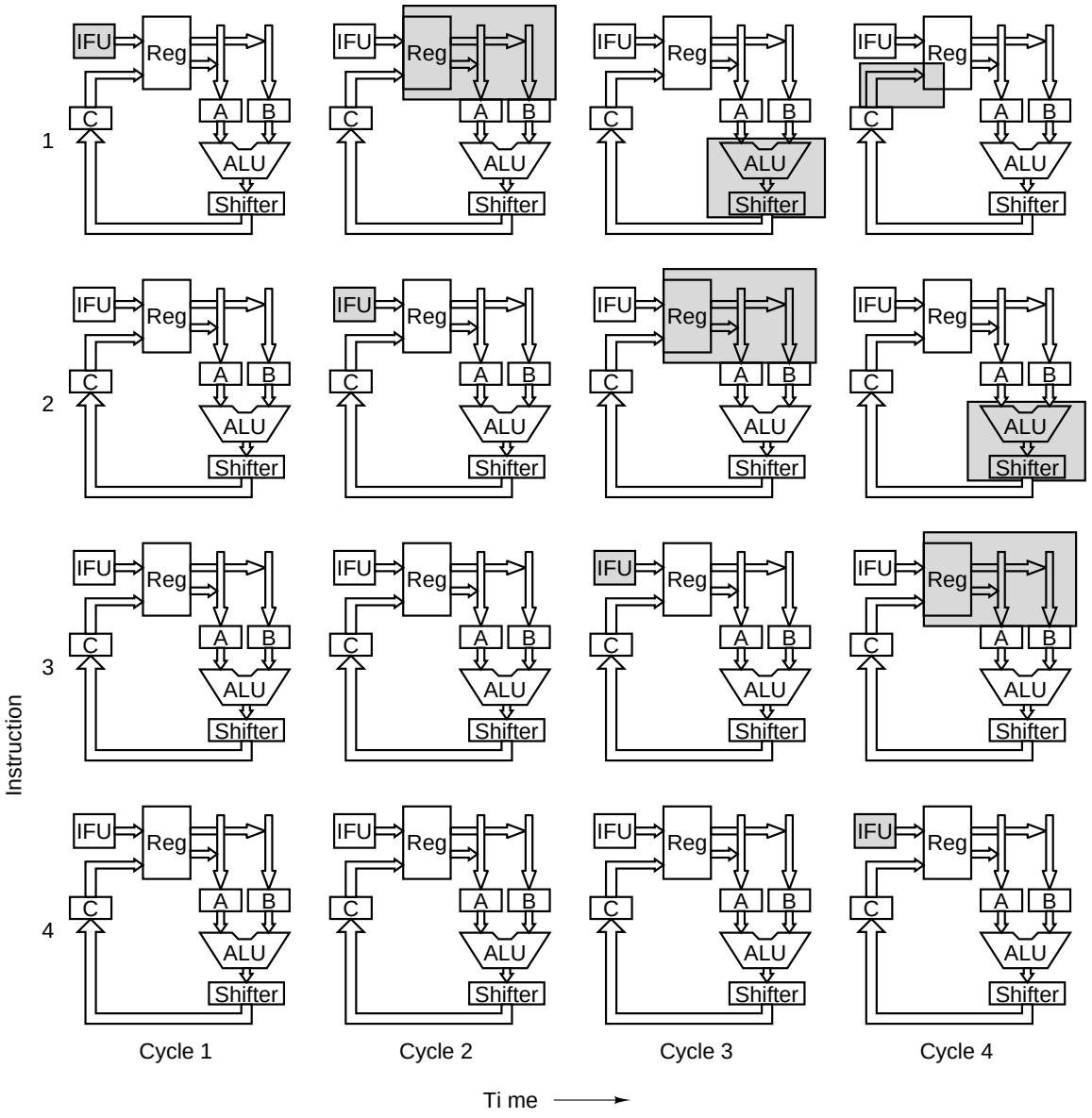


Figure 4-34. Graphical illustration of how a pipeline works.

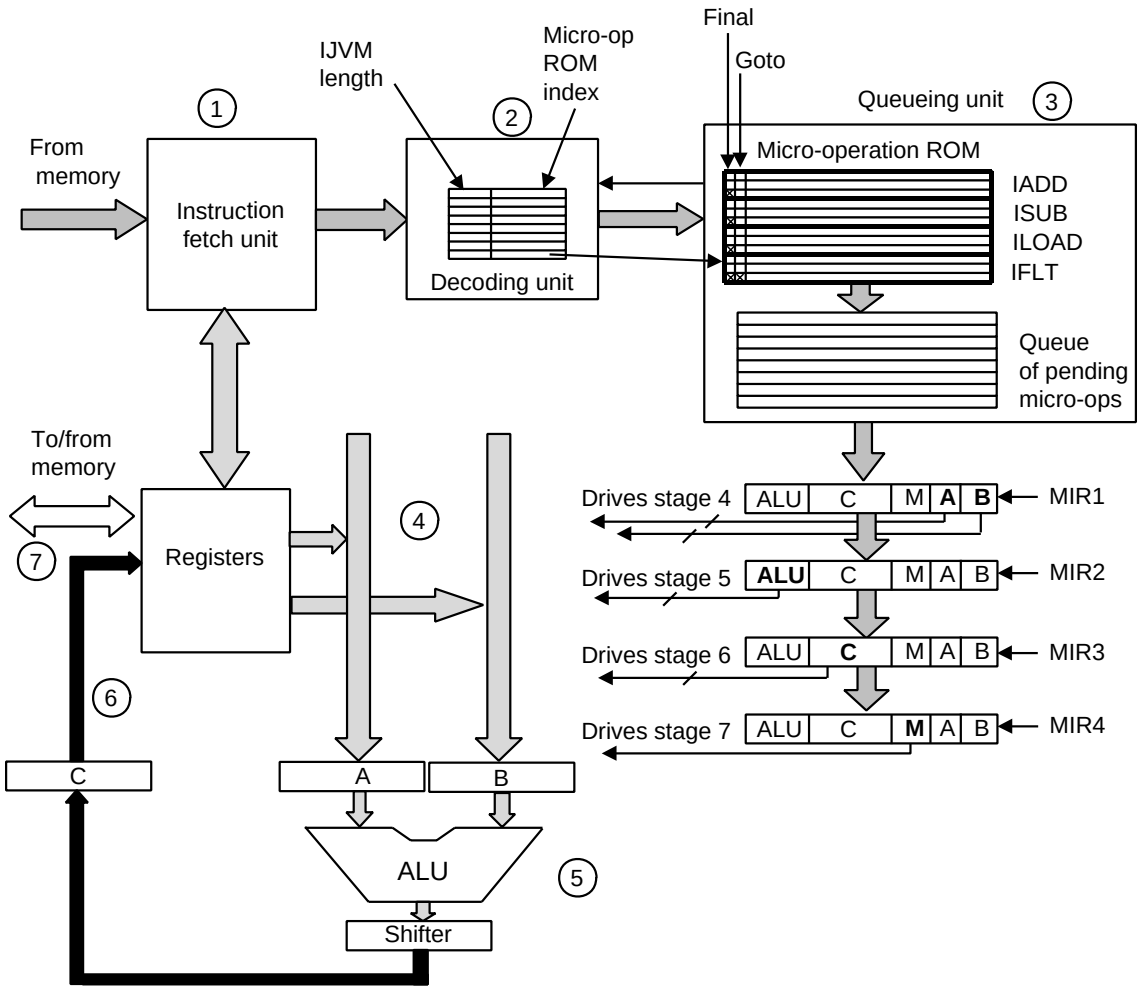


Figure 4-35. The main components of the Mic-4.

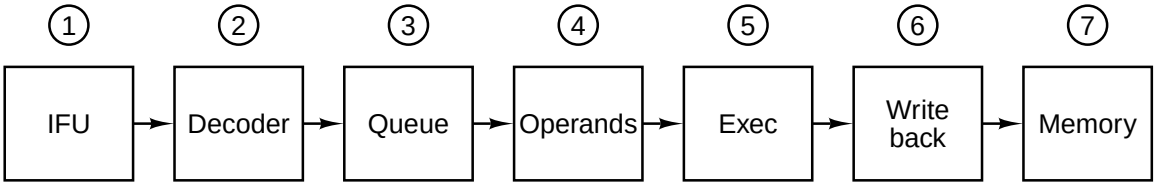


Figure 4-36. The Mic-4 pipeline.

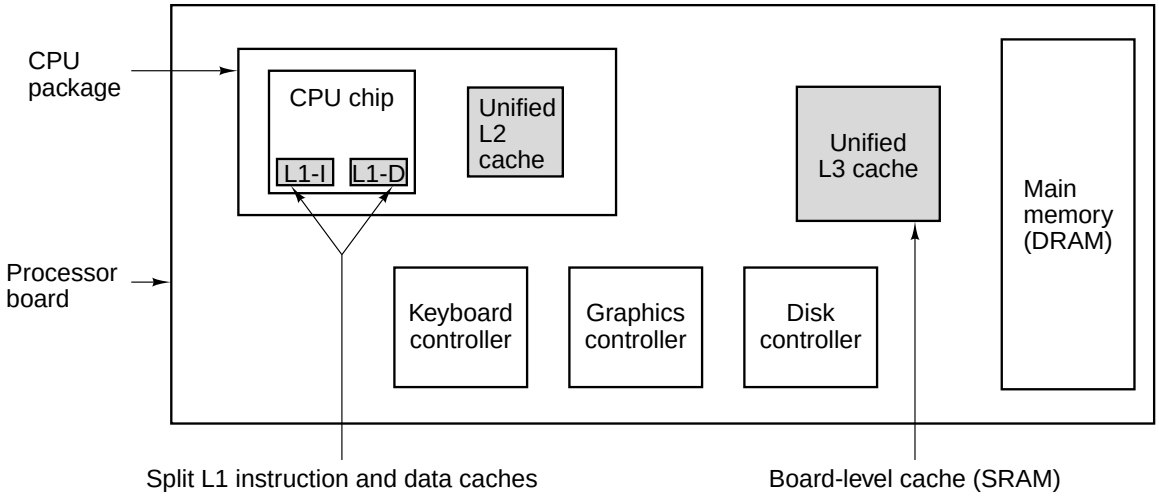


Figure 4-37. A system with three levels of cache.

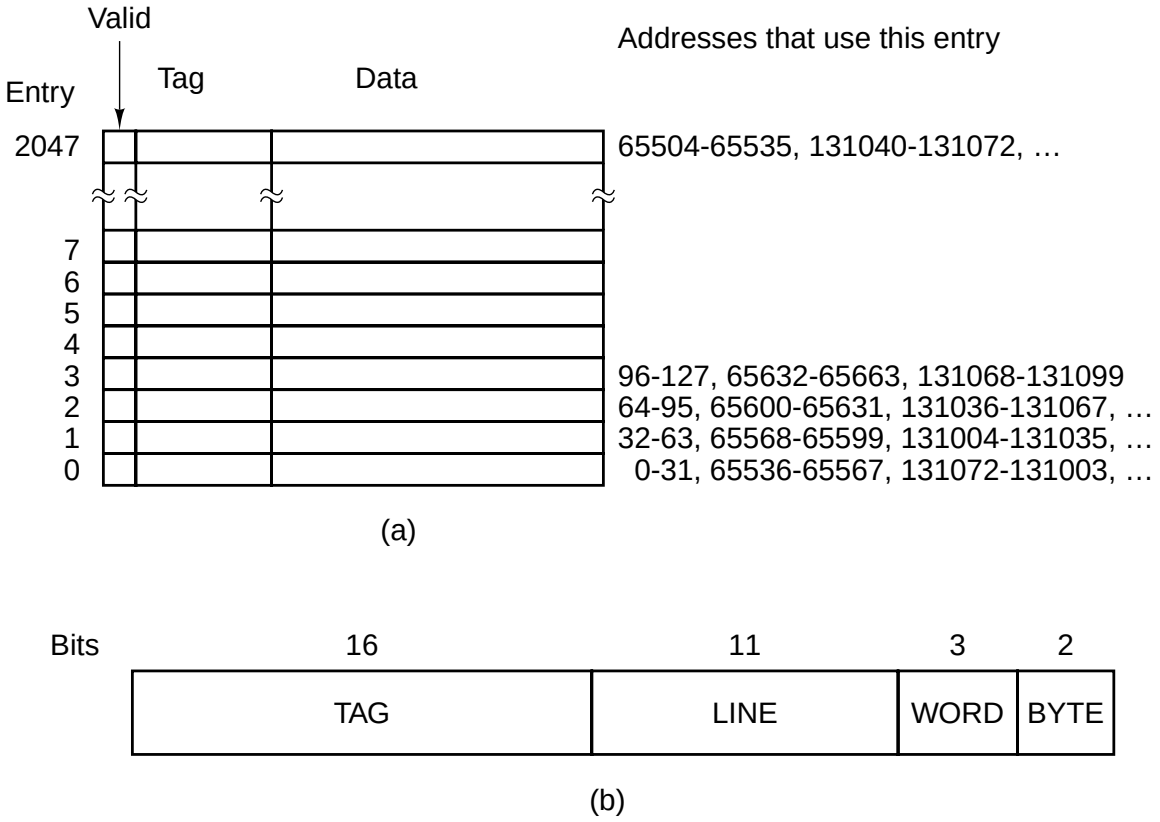


Figure 4-38. (a) A direct-mapped cache. (b) A 32-bit virtual address.

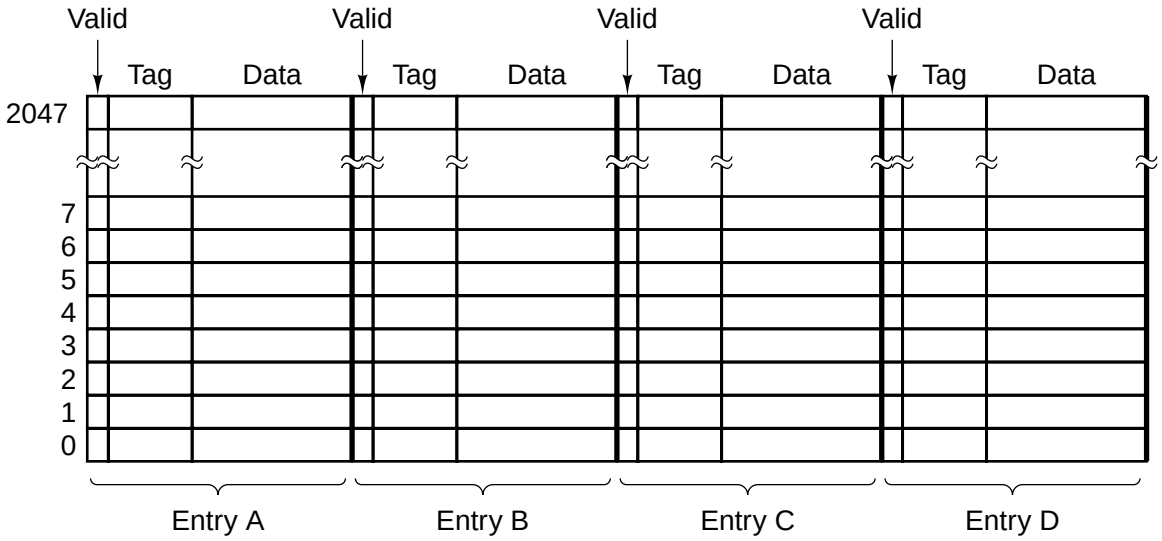


Figure 4-39. A four-way associative cache.

```
if (i == 0)                    CMP i,0; compare i to 0
    k = 1;                     BNE Else; branch to Else if not equal
else                           Then:   MOV k,1; move 1 to k
    k = 2;                     BR Next; unconditional branch to Next
                          Else: MOV k,2; move 2 to k
                          Next:
```

(a) (b)

Figure 4-40. (a) A program fragment. (b) Its translation to a generic assembly language.

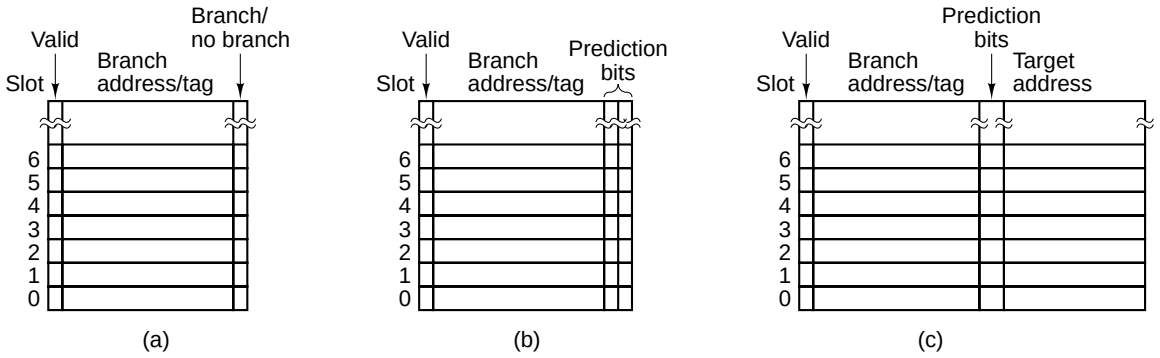


Figure 4-41. (a) A 1-bit branch history. (b) A 2-bit branch history. (c) A mapping between branch instruction address and target address.

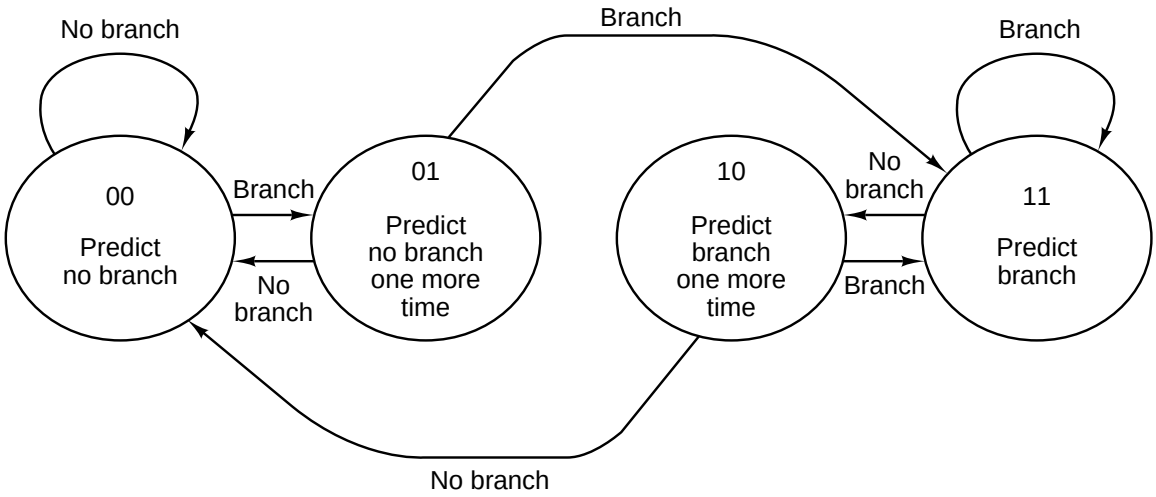


Figure 4-42. A 2-bit finite-state machine for branch prediction.

					Registers being read								Registers being written							
Cy	**#**	**Decoded**	**Iss**	**Ret**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	–		3	2	1									1	1	1		
3					3	2	1									1	1	1		
4				1	2	1	1										1	1		
				2	1	1												1		
				3																
5			4			1			1										1	
	5	R7=R1*R2	5			2	1		1										1	1
6	6	R1=R0–R2	–			2	1		1										1	1
7				4		1	1													1
8				5																
9			6		1		1							1						
	7	R3=R3*R1	7		1	1	1	1						1		1				
10					1	1	1	1						1		1				
11				6		1		1								1				
12				7																
13	8	R1=R4+R4	8						2					1						
14									2					1						
15				8																

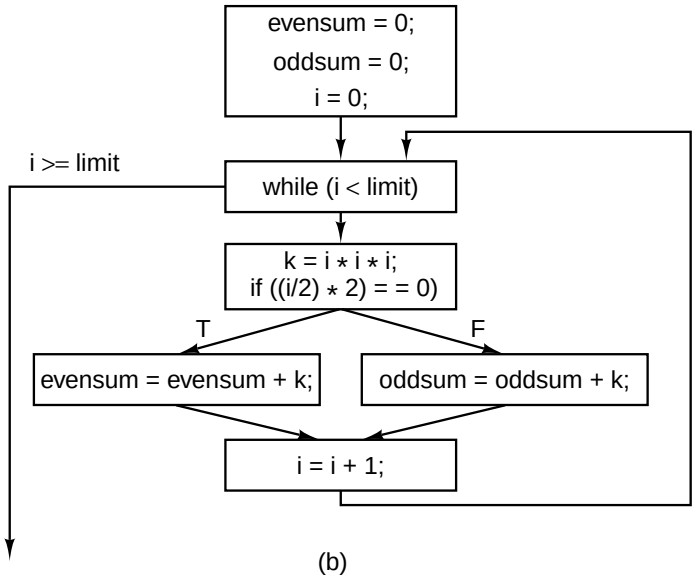
Figure 4-43. Operation of a superscalar CPU with in-order issue and in-order completion.

					Registers being read								Registers being written							
Cy	**#**	**Decoded**	**Iss**	**Ret**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	–		3	2	1									1	1	1		
3	5	R7=R1*R2	5		3	3	2									1	1	1		1
	6	S1=R0–R2	6		4	3	3									1	1	1		1
				2	3	3	2									1		1		1
4			4		3	4	2		1							1		1	1	1
	7	R3=R3*S1	–		3	4	2		1							1		1	1	1
	8	S2=R4+R4	8		3	4	2		3							1		1	1	1
				1	2	3	2		3									1	1	1
				3	1	2	2		3										1	1
5				6		2	1		3					1					1	1
6			7			2	1	1	3					1		1			1	1
				4		1	1	1	2					1		1				1
				5				1	2					1		1				
				8				1								1				
7								1								1				
8								1								1				
9				7																

Figure 4-44. Operation of a superscalar CPU with out-of-order issue and out-of-order completion.

```
evensum = 0;
oddsum = 0;
i = 0;
while (i < limit) {
    k = i * i * i;
    if ((i/2) * 2) == 0)
        evensum = evensum + k;
    else
        oddsum = oddsum + k;
    i = i + 1;
}
```

(a)



(b)

Figure 4-45. (a) A program fragment. (b) The corresponding basic block graph.

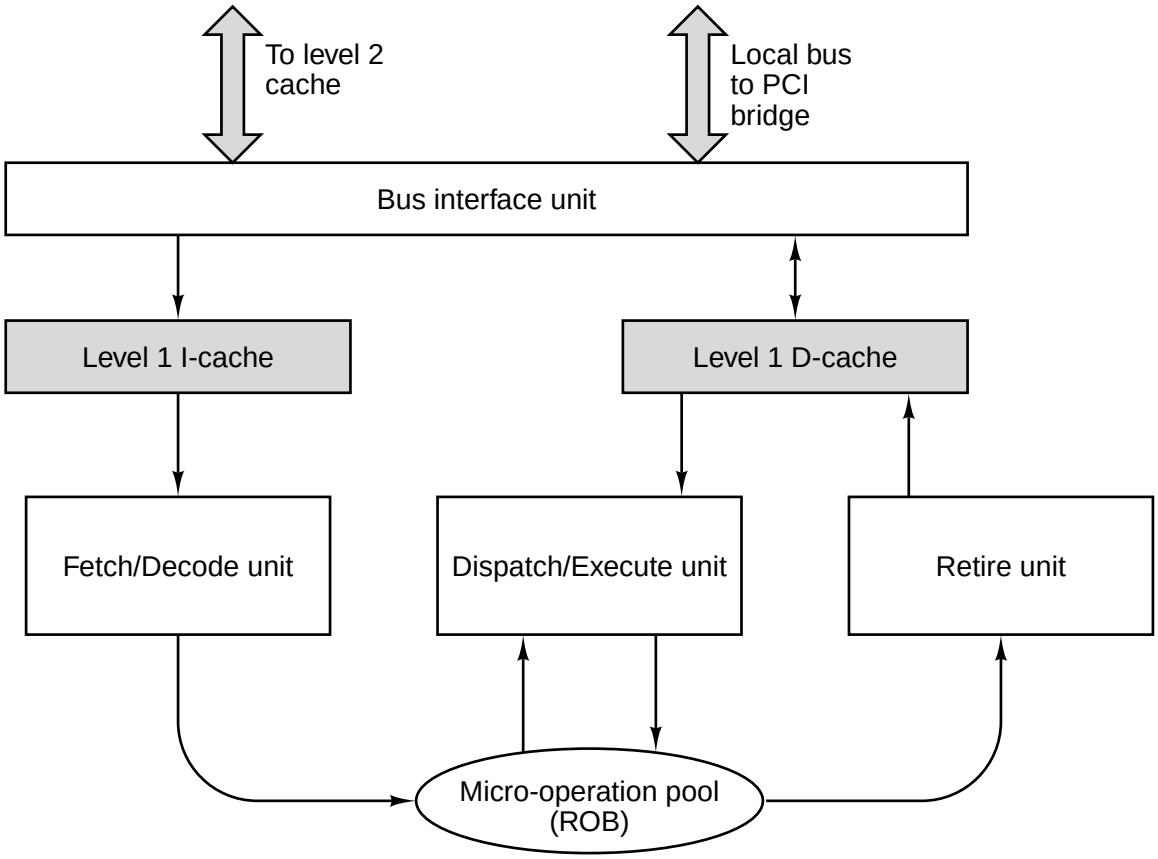


Figure 4-46. The Pentium II microarchitecture.

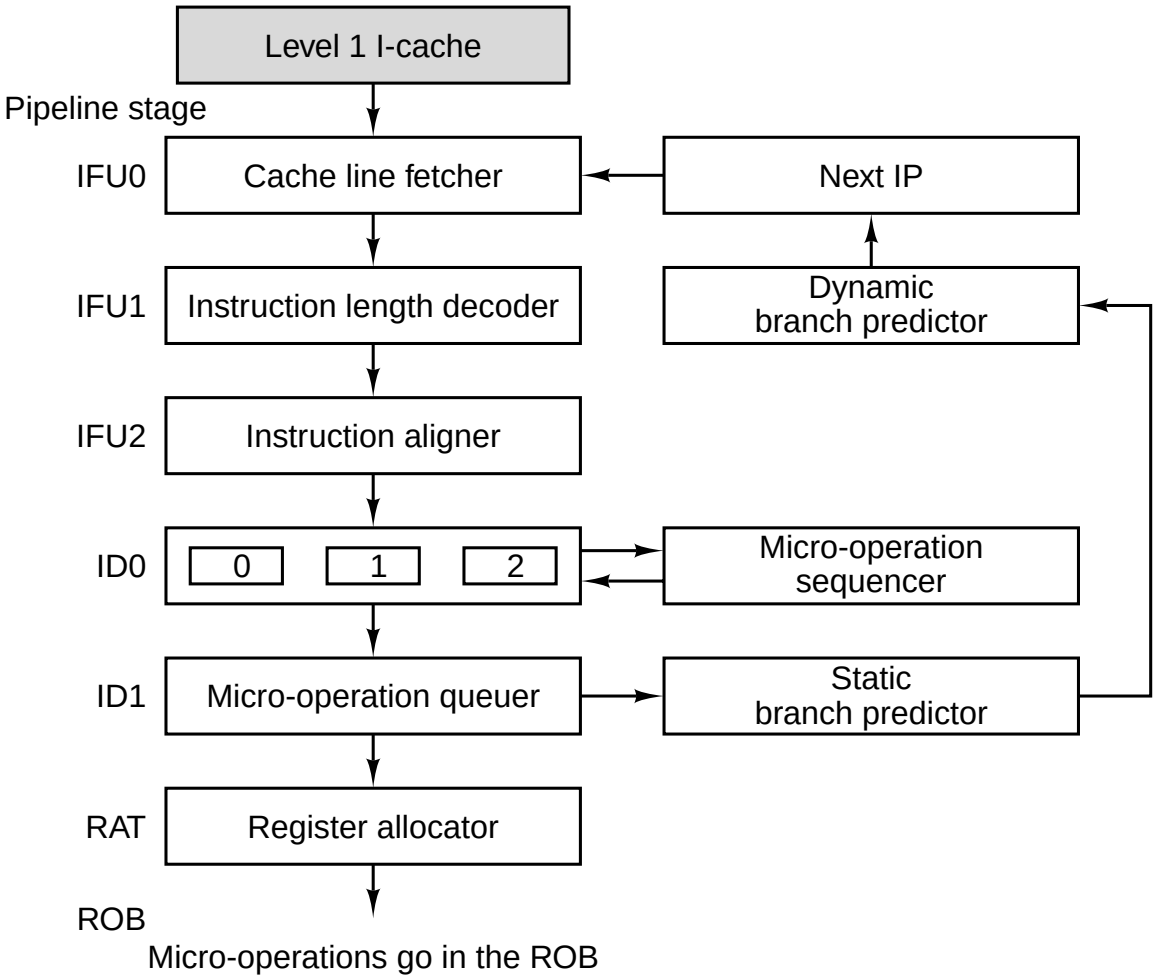


Figure 4-47. Internal structure of the Fetch/Decode unit (simplified).

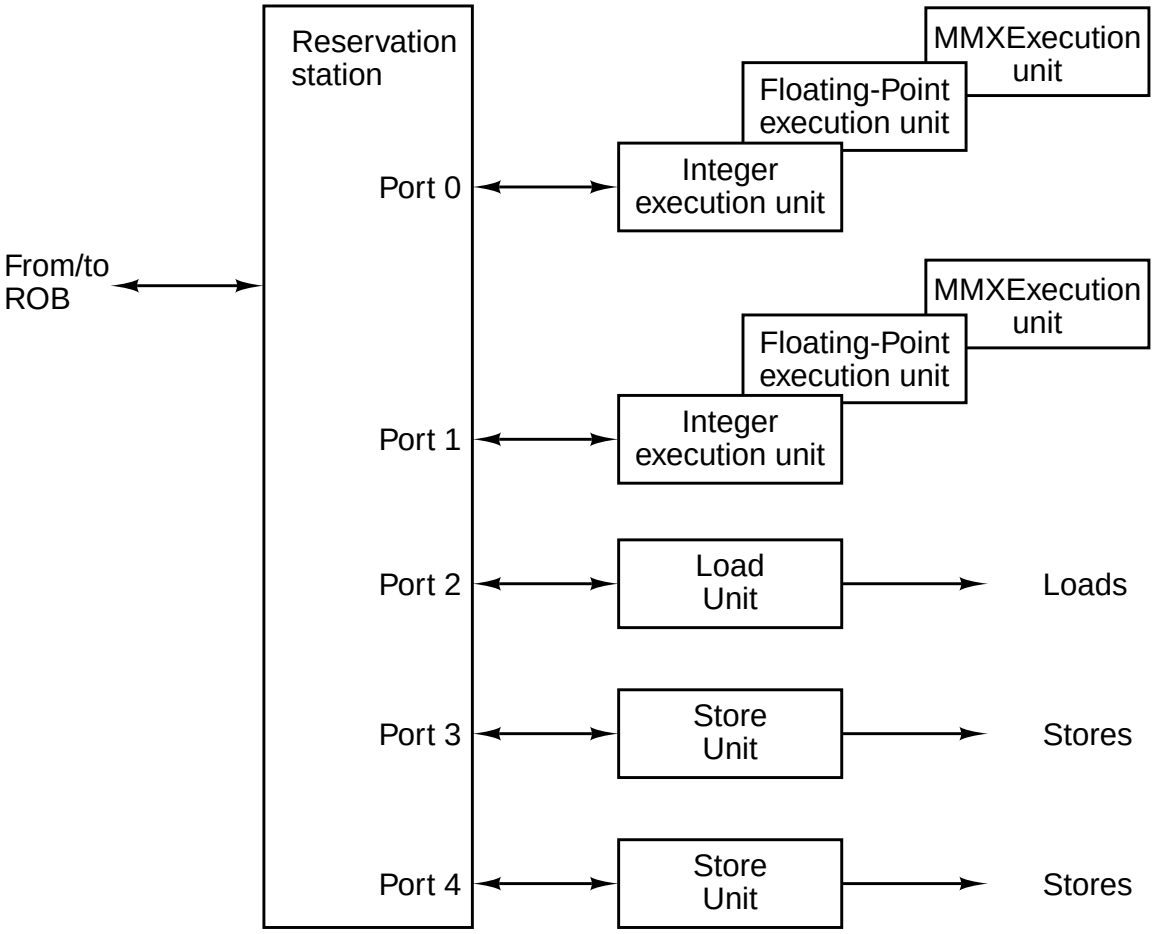


Figure 4-48. The Dispatch/Execute unit.

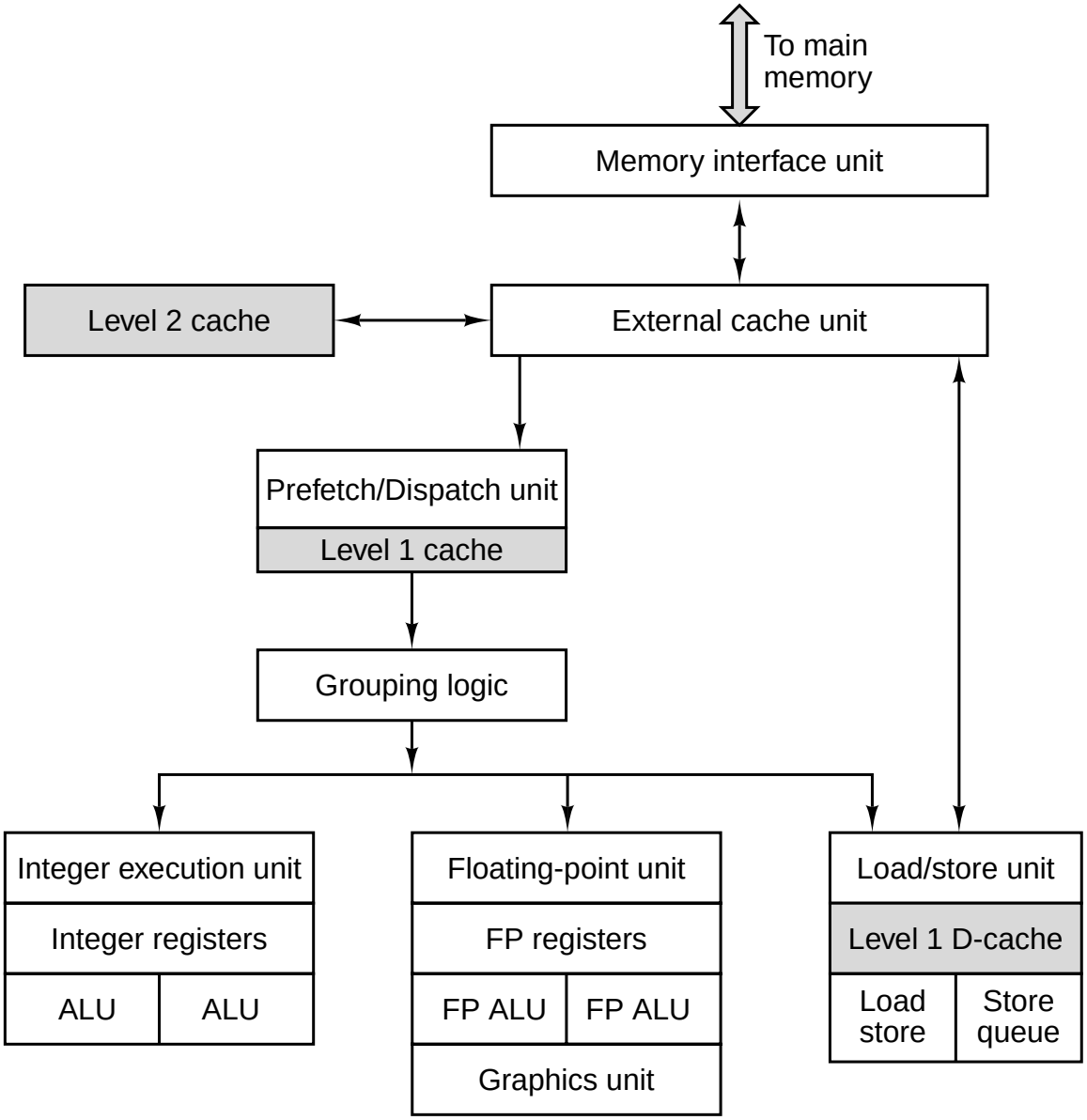


Figure 4-49. The UltraSPARC II microarchitecture.

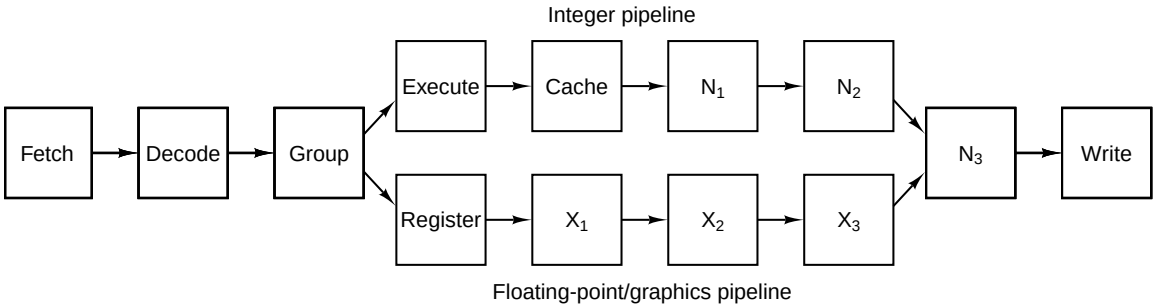


Figure 4-50. The UltraSPARC II's pipeline.

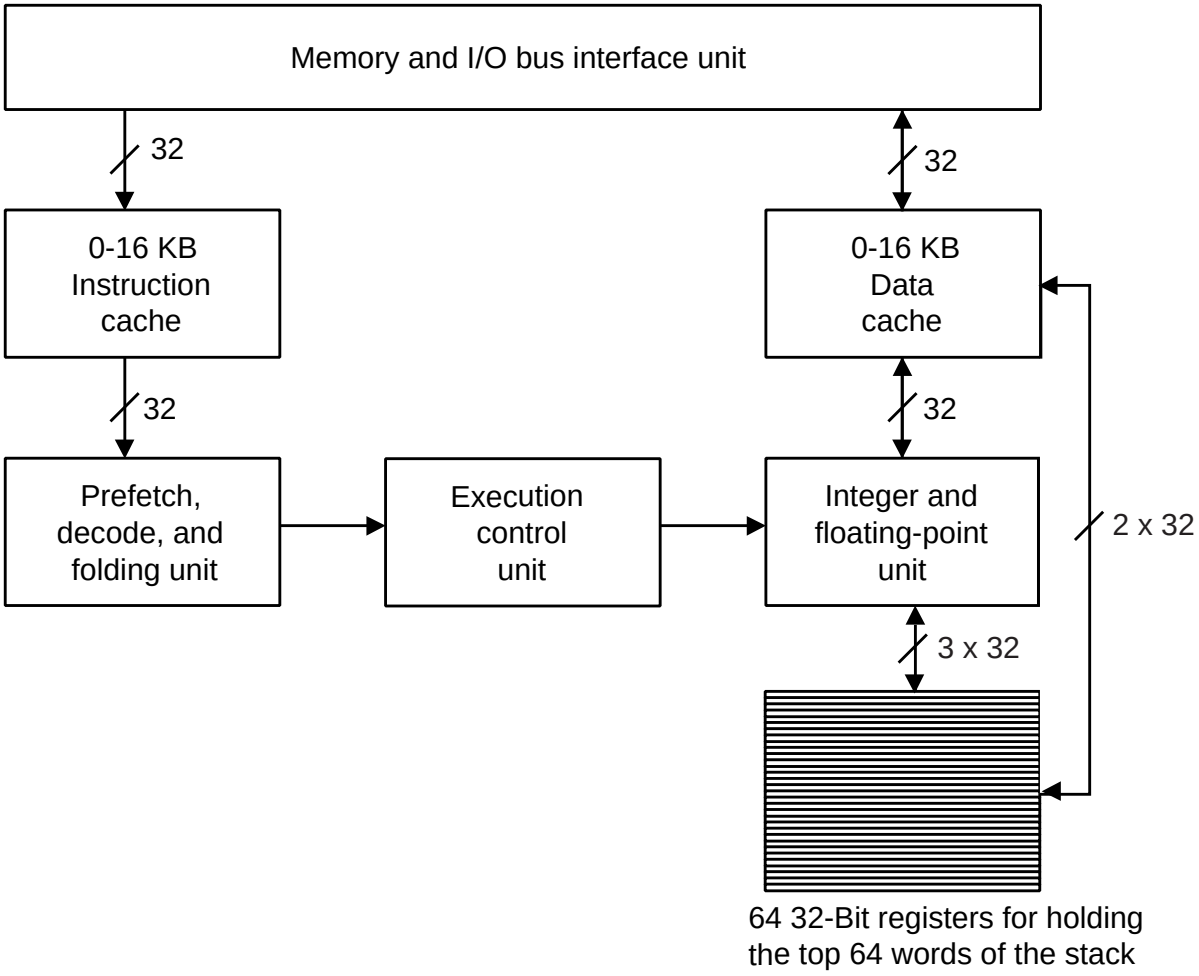


Figure 4-51. The block diagram of the picoJava II with both level 1 caches and the floating-point unit. This is configuration of the microJava 701.

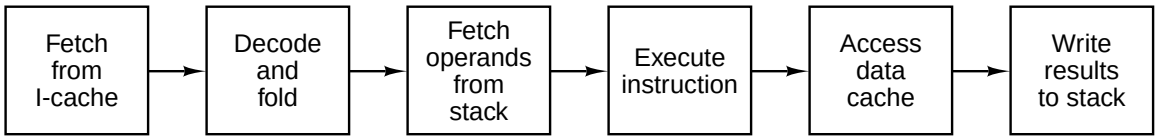


Figure 4-52. The picoJava II has a six-stage pipeline.

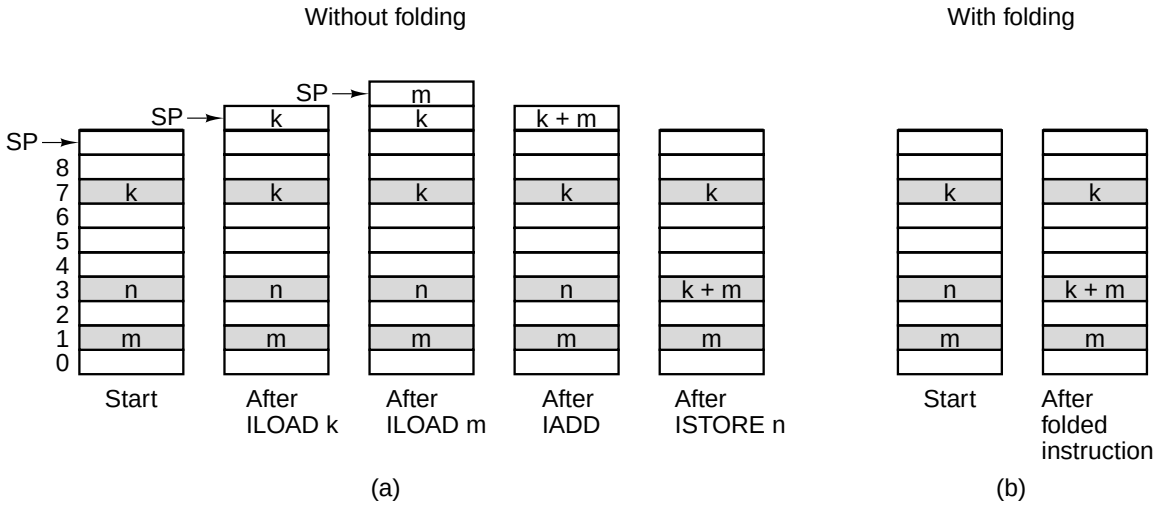


Figure 4-53. (a) Execution of a four-instruction sequence to compute n = k + m. (b) The same sequence folded to one instruction.

Group	Description	Example
NF	Nonfoldable instructions	GOTO
LV	Pushing a word onto the stack	ILOAD
MEM	Popping a word and storing it in memory	ISTORE
BG1	Operations using one stack operand	IFEQ
BG2	Operations using two stack operands	IF_CMPEQ
OP	Computations on two operands with one result	IADD

Figure 4-54. JVM instruction groups for folding purposes.

Instruction sequence				Example
LV	LV	OP	MEM	ILOAD, ILOAD, IADD, ISTORE
LV	LV	OP		ILOAD, ILOAD, IADD
LV	LV	BG2		ILOAD, ILOAD, IF_CMPEQ
LV	BG1			ILOAD, IFEQ
LV	BG2			ILOAD, IF_CMPEQ
LV	MEM			ILOAD, ISTORE
OP	MEM			IADD, ISTORE

Figure 4-55. Some of the JVM instruction sequences that can be folded.

4.5 IMPROVING PERFORMANCE

All computer manufacturers want their systems to run as fast as possible. In this section, we will look at a number of advanced techniques currently being investigated to improve system (primarily CPU and memory) performance. Due to the highly competitive nature of the computer industry, the lag between new research ideas that can make a computer faster and their incorporation into products is surprisingly short. Consequently, most of the ideas we will discuss are already in use in a wide variety of existing products.

The ideas to be discussed fall into two rough categories: implementation improvements and architectural improvements. Implementation improvements are ways of building a new CPU or memory to make the system run faster without changing the architecture. Modifying the implementation without changing the architecture means that old programs will run on the new machine, a major selling point. One way to improve the implementation is to use a faster clock, but this is not the only way. The performance gains from the 80386 through the 80486, Pentium, and Pentium Pro, to the Pentium II are due to better implementations, as the architecture has remained essentially the same through all of them.

Some kinds of improvements can be made only by changing the architecture. Sometimes these changes are incremental, such as adding new instructions or registers, so that old programs will continue to run on the new models. In this case, to get the full performance, the software must be changed, or at least recompiled with a new compiler that takes advantage of the new features.

However, once in a few decades, designers realize that the old architecture has outlived its usefulness and that the only way to make progress is start all over again. The RISC revolution in the 1980s was one such breakthrough; another one is in the air now. We will look at one example (the Intel IA-64) in Chap. 5.

In the rest of this section we will look at four different techniques for improving CPU performance. We will start with three well-established implementation improvements and then move on to one that needs a little architectural support to work best. These techniques are cache memory, branch prediction, out-of-order execution with register renaming, and speculative execution.

4.5.1 Cache Memory

One of the most challenging aspects of computer design throughout history has been to provide a memory system able to provide operands to the processor at the speed it can process them. The recent high rate of growth in processor speed has not been accompanied by a corresponding speedup in memories. Relative to CPUs, memories have been getting slower for decades. Given the enormous importance of primary memory, this situation has greatly limited the development of high-performance systems, and has stimulated research on ways to get around the problem of memory speeds that are much slower than CPU speeds, an

relatively speaking, getting worse every year.

Modern processors place overwhelming demands on a memory system, both in terms of latency (the delay in supplying an operand) and bandwidth (the amount of data supplied per unit of time). Unfortunately, these two aspects of a memory system are largely at odds. Many techniques for increasing bandwidth do so only by increasing latency. For example, the pipelining techniques used in the Mic-3 can be applied to a memory system, with multiple, overlapping memory requests handled efficiently. Unfortunately, as with the Mic-3, this results in greater latency for individual memory operations. As processor clock speeds get faster, it becomes more and more difficult to provide a memory system capable of supplying operands in one or two clock cycles.

One way to attack this problem is by providing caches. As we saw in Sec. 2.2.5, a cache holds the most recently used memory words in a small, fast memory, speeding up access to them. If a large enough percentage of the memory words needed are in the cache, the effective memory latency can be reduced enormously.

One of the most effective techniques for improving both bandwidth and latency comes from the use of multiple caches. A basic technique that works very effectively is to introduce a separate cache for instructions and data. There are several benefits from having separate caches for instructions and data, often called a **split cache**. First, memory operations can be initiated independently in each cache, effectively doubling the bandwidth of the memory system. This is the reason that it makes sense to provide two separate memory ports, as we did in the Mic-1: each port has its own cache. Note that each cache has independent access to the main memory.

Today, many memory systems are more complicated than this, and an additional cache, called a **level 2 cache**, may reside between the instruction and data caches and main memory. In fact, there may be three or more levels of cache as more sophisticated memory systems are required. In Fig. 4-1 we see a system with three levels of cache. The CPU chip itself contains a small instruction cache and a small data cache, typically 16 KB to 64 KB. Then there is the level 2 cache, which is not on the CPU chip, but may be included in the CPU package, next to the CPU chip and connected to it by a high-speed path. This cache is generally unified, containing a mixture of data and instructions. A typical size for the L2 cache is 512 KB to 1 MB. The third-level cache is on the processor board and consists of a few megabytes of SRAM, which is much faster than the main DRAM memory. Caches are generally inclusive, with the full contents of the level 1 cache being in the level 2 cache and the full contents of the level 2 cache being in the level 3 cache.

Caches depend on two kinds of address locality to achieve their goal. **Spatial locality** is the observation that memory locations with addresses numerically similar to a recently accessed memory location are likely to be accessed in the near future. Caches exploit this property by bringing in more data than have been

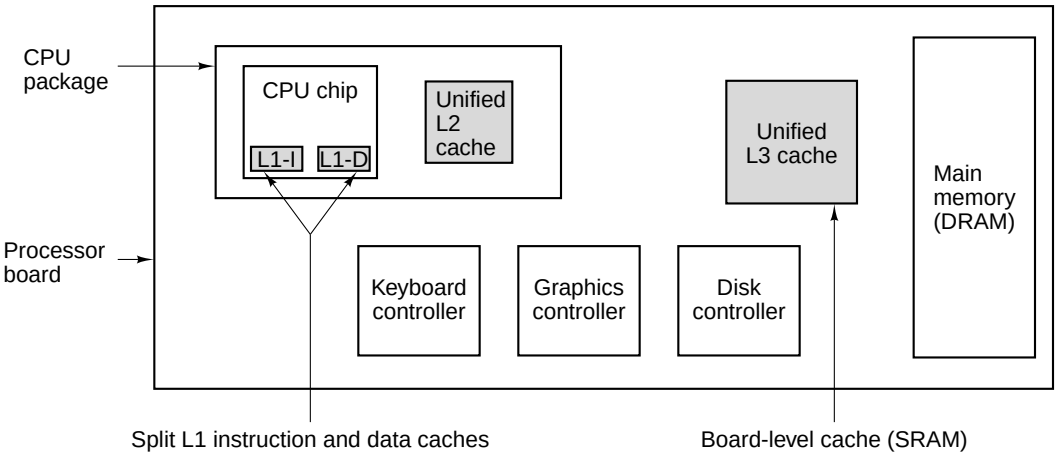


Figure 4-1. A system with three levels of cache.

requested, with the expectation that future requests can be anticipated. **Temporal locality** occurs when recently accessed memory locations are accessed again. This may occur, for example, to memory locations near the top of the stack, or instructions inside a loop. Temporal locality is exploited in cache designs primarily by the choice of what to discard on a cache miss. Many cache replacement algorithms exploit temporal locality by discarding those entries that have not been recently accessed.

All caches use the following model. Main memory is divided up into fixed-size blocks called **cache lines**. A cache line typically consists of 4 to 64 consecutive bytes. Lines are numbered consecutively starting at 0, so with a 32-byte line size, line 0 is bytes 0 to 31, line 1 is bytes 32 to 63, and so on. At any instant, some lines are in the cache. When memory is referenced, the cache controller circuit checks to see if the word referenced is currently in the cache. If so, the value there can be used, saving a trip to main memory. If the word is not there, some line entry is removed from the cache and the line needed is fetched from memory or lower level cache to replace it. Many variations on this scheme exist, but in all of them the idea is to keep the most heavily-used lines in the cache as much as possible, to maximize the number of memory references satisfied out of the cache.

Direct-Mapped Caches

The simplest cache is known as a **direct-mapped cache**. An example single-level direct-mapped cache is shown in Fig. 4-2(a). This example cache contains 2048 entries. Each entry (row) in the cache can hold exactly one cache line from main memory. With a 32-byte cache line size (for this example), the

cache can hold 64 KB. Each cache entry consists of three parts:

1. The Valid bit indicates whether there is any valid data in this entry or not. When the system is booted (started), all entries are marked as invalid.
2. The Tag field consists of a unique, 16-bit value identifying the corresponding line of memory from which the data came.
3. The Data field contains a copy of the data in memory. This field holds one cache line of 32 bytes.

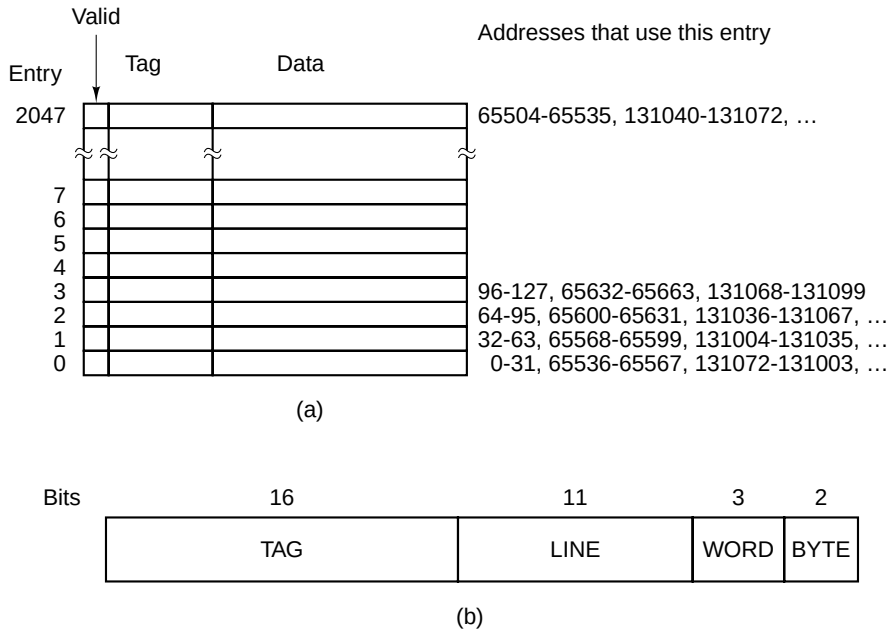


Figure 4-2. (a) A direct-mapped cache. (b) A 32-bit virtual address.

In a direct-mapped cache, a given memory word can be stored in exactly one place within the cache. Given a memory address, there is only one place to look for it in the cache. If it is not there, then it is not in the cache. For storing and retrieving data from the cache, the address is broken into four components, as shown in Fig. 4-2(b):

1. The TAG field corresponds to the Tag bits stored in a cache entry.
2. The LINE field indicates which cache entry holds the corresponding data, if they are present.
3. The WORD field tells which word within a line is referenced.
4. The BYTE field is usually not used, but if only a single byte is

requested, it tells which byte within the word is needed. For a cache supplying only 32-bit words, this field will always be 0.

When the CPU produces a memory address, the hardware extracts the 11 LINE bits from the address and uses these to index into the cache to find one of the 2048 entries. If that entry is valid, the TAG field of the memory address and the Tag field in cache entry are compared. If they agree, the cache entry holds the word being requested, a situation called a **cache hit**. On a hit, a word being read can be taken from the cache, eliminating the need to go to memory. Only the word actually needed is extracted from the cache entry. The rest of the entry is not used. If the cache entry is invalid or the tags do not match, the needed entry is not present in the cache, a situation called a **cache miss**. In this case, the 32-byte cache line is fetched from memory and stored in the cache entry, replacing what was there. However, if the existing cache entry has been modified since being loaded, it must be written back to main memory before being discarded.

Despite the complexity of the decision, access to a needed word can be remarkably fast. As soon as the address is known, the exact location of the word is known *if it is present in the cache*. This means that it is possible to read the word out of the cache and deliver it to the processor at the same time that it is being determined if this is the correct word (by comparing tags). So the processor actually receives a word from the cache simultaneously, or possibly even before it knows whether the word is the requested one.

This mapping scheme puts consecutive memory lines in consecutive cache entries, In fact, up to 64K bytes of contiguous data can be stored in the cache. However, two lines that differ in their address by precisely 64K (65,536 bytes) or any integral multiple of that number cannot be stored in the cache at the same time (because they have the same LINE value). For example, if a program accesses data at location X and next executes an instruction that needs data at location X + 65,536 (or any of the other location within the same line), the second instruction will force the cache entry to be reloaded, overwriting what was there. If this happens often enough, it can result in poor behavior. In fact, the worst-case behavior of a cache is worse than if there were no cache at all, since each memory operation involves reading in an entire cache line instead of just one word.

Direct-mapped caches are the most common kind of cache, and they perform quite effectively, because collisions such as the one described above can be made to occur only rarely, or not at all. For example, a very clever compiler can take cache collisions into account when placing instructions and data in memory. Notice that the particular case described would not occur in a system with separate instruction and data caches, because the colliding requests would be serviced by different caches. Thus we see a second benefit of two caches rather than one: more flexibility in dealing with conflicting memory patterns.

Set-Associative Caches

As mentioned above, many different lines in memory compete for the same cache slots. If a program using the cache of Fig. 4-2(a) heavily uses words at addresses 0 and at 65,536, there will be constant conflicts, with each reference potentially evicting the other one from the cache. A solution to this problem is to allow two or more lines in each cache entry. A cache with *n* possible entries for each address is called an **n-way set-associative cache**. A four-way associative cache is illustrated in Fig. 4-3.

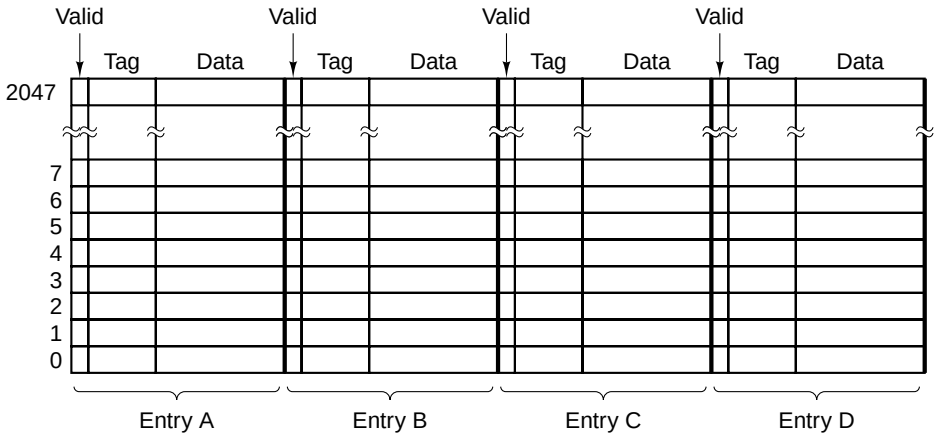


Figure 4-3. A four-way associative cache.

A set-associative cache is inherently more complicated than a direct-mapped cache because although the correct cache entry to examine can be computed from the memory address being referenced, a set of *n* cache entries must be checked to see if the needed line is present. Nevertheless, experience shows that two-way and four-way caches perform well enough to make this extra circuitry worthwhile.

The use of a set-associative cache presents the designer with a choice. When a new entry is to be brought into the cache, which of the present items should be discarded? The optimal decision, of course, requires a peek into the future, but a pretty good algorithm for most purposes is **LRU** (**Least Recently Used**). This algorithm keeps an ordering of each set of locations that could be accessed from a given memory location. Whenever any of the present lines are accessed, it updates the list, marking that entry the most recently accessed. When it comes time to replace an entry, the one at the end of the list—the least recently accessed—is the one discarded.

Carried to the extreme, a 2048-way cache containing a single set of 2048 line entries is also possible. Here all memory addresses map onto the single set, so the lookup requires comparing the address against all 2048 tags in the cache. Note that each entry must now have tag-matching logic. Since the LINE field is of 0

length, the TAG field is the entire address except for the WORD and BYTE fields. Furthermore, when a cache line is replaced, all 2048 locations are possible candidates for replacement. Maintaining an ordered list of 2048 entries requires a great deal of bookkeeping, making LRU replacement infeasible. (Remember that this list has to be updated on every memory operation, not just on a miss). Surprisingly, high-associativity caches do not improve performance much over low-associativity caches under most circumstances, and in some cases actually perform worse. For these reasons, associativity beyond four-way is quite unusual.

Finally, writes pose a special problem for caches. When a processor writes a word, and the word is in the cache, it obviously must either update the word or discard the cache entry. Nearly all designs update the cache. But what about updating the copy in main memory? This operation can be deferred until later, when the cache line is ready to be replaced by the LRU algorithm. This choice is difficult, and neither option is clearly preferable. Immediately updating the entry in main memory is referred to as **write through**. This approach is generally simpler to implement and more reliable, since the memory is always up to date—helpful, for example, if an error occurs and it is necessary to recover the state of the memory. Unfortunately, it also usually requires more write traffic to memory, so more sophisticated implementations tend to employ the alternative, known as **write deferred**, or **write back**.

A related problem must be addressed for writes: what if a write occurs to a location that is not currently cached? Should the data be brought into the cache, or just written out to memory? Again, neither answer is always best. Most designs that defer writes to memory tend to bring data into the cache on a write miss, a technique known as **write allocation**. Most designs employing write through, on the other hand, tend not to allocate an entry on a write because this option complicates an otherwise simple design. Write allocation wins only if there are repeated writes to the same or different words within a cache line.

4.5.2 Branch Prediction

Modern computers are highly pipelined. The pipeline of Fig. 4-0 has seven stages; high-end computers sometimes have 10-stage pipelines or even more. Pipelining works best on linear code, so the fetch unit can just read in consecutive words from memory and send them off to the decode unit in advance of their being needed.

The only minor problem with this wonderful model is that it is not the slightest bit realistic. Programs are not linear code sequences. They are full of branch instructions. Consider the simple statements of Fig. 4-4(a). A variable, *i*, is compared to 0 (probably the most common test in practice). Depending on the result, another variable, *k*, gets assigned one of two possible values.

A possible translation to assembly language is shown in Fig. 4-4(b). We will study assembly language later in this book, and the details are not important now,

```
if (i == 0)                          CMP i,0     ; compare i to 0
    k = 1;                           BNE Else    ; branch to Else if not equal
else                        Then:    MOV k,1     ; move 1 to k
    k = 2;                           BR Next     ; unconditional branch to Next
                                Else: MOV k,2     ; move 2 to k
                                Next:
    (a)                                  (b)
```

Figure 4-4. (a) A program fragment. (b) Its translation to a generic assembly language.

but depending on the machine and the compiler, code more-or-less like that of Fig. 4-4(b) is likely. The first instruction compares *i* to 0. The second one branches to the label *Else* (the start of the else clause) if *i* is not 0. The third instruction assigns 1 to *k*. The fourth instruction branches to the code for the next statement. The compiler has conveniently planted a label, *Next*, there, so there is a place to branch to. The fifth instruction assigns 2 to *k*.

The thing to observe here is that two of the five instructions are branches. Furthermore, one of these, BNE, is a conditional branch (a branch that is taken if and only if some condition is met, in this case, that the two operands in the previous CMP are equal). The longest linear code sequence here is two instructions. As a consequence, fetching instructions at a high rate to feed the pipeline is very difficult.

At first glance, it might appear that unconditional branches, such as the instruction BR Next in Fig. 4-4(b), are not a problem. After all, there is no ambiguity about where to go. Why can't the fetch unit just continue to read instructions from the target address (the place that will be branched to)?

The trouble lies in the nature of pipelining. In Fig. 4-0, for example, we see that instruction decoding occurs in the second stage. Thus the fetch unit has to decide where to fetch from next before it knows what kind of instruction it just got. Only one cycle later can it learn that it just picked up an unconditional branch, and by then it has already started to fetch the instruction following the unconditional branch. As a consequence, a substantial number of pipelined machines (such as the UltraSPARC II) have the property that the instruction *following* an unconditional branch is executed, even though logically it should not be. The position after a branch is called a **delay slot**. The Pentium II [and the machine used in Fig. 4-4(b)] do not have this property, but the internal complexity to get around this problem is often enormous. An optimizing compiler will try to find some useful instruction to put in the delay slot, but frequently there is nothing available, so it is forced to insert a NOP instruction there. Doing so keeps the program correct, but makes it bigger and slower.

Annoying as unconditional branches are, conditional branches are worse. Not only do they also have delay slots, but now the fetch unit does not where to read

from until much later in the pipeline. Early pipelined machines just **stalled** until it was known whether the branch would be taken or not. Stalling for three or four cycles on every conditional branch, especially if 20% of the instructions are conditional branches, wreaks havoc with the performance.

Consequently, what most machines do when they hit a conditional branch is predict whether it is going to be taken or not. It would be nice if we could just plug a crystal ball into a free PCI slot to help out with the prediction, but so far this approach has not borne fruit.

Lacking such a peripheral, various ways have been devised to do the prediction. One very simple way is as follows: assume that all backward conditional branches will be taken and that all forward ones will not be taken. The reasoning behind the first part is that backward branches are frequently located at the end of a loop. Most loops are executed multiple times, so guessing that a branch back to the top of the loop will be taken is generally a good bet.

The second part is shakier. Some forward branches occur when error conditions are detected in software (e.g., a file cannot be opened). Errors are rare, so most of the branches associated with them are not taken. Of course, there are plenty of forward branches not related to error handling, so the success rate is not nearly as good as with backward branches. While not fantastic, this rule is at least better than nothing.

If a branch is correctly predicted, there is nothing special to do. Execution just continues at the target address. The trouble comes when a branch is predicted wrongly. Figuring out where to go and going there is not difficult. The hard part is undoing instructions that have already been executed and should not have been.

There are two ways of going about this. The first way is to allow instructions fetched after a predicted conditional branch to execute until they try to change the machine's state (e.g., storing into a register). Instead of overwriting the register, the value computed is put into a (secret) scratch register and only copied to the real register after it is known that the prediction was correct. The second way is to record the value of any register about to be overwritten (e.g., in a secret scratch register), so the machine can be rolled back to the state it had at the time of the mispredicted branch. Both solutions are complex and require industrial-strength bookkeeping to get them right. And if a second conditional branch is hit before it is known whether the first one was predicted right, things can get really messy.

Dynamic Branch Prediction

Clearly, having the predictions be accurate is of great value, since it allows the CPU to proceed at full speed. As a consequence, a great deal of current research aims at improving branch prediction algorithms (e.g., Driesen and Holzle, 1998; Juan et al., 1998; Pan et al., 1992; Sechrest et al., 1996; Sprangle et al., 1997; and Yeh and Patt, 1991). One approach is for the CPU to maintain a history table (in special hardware), in which it logs conditional branches as they occur, so

they can be looked up when they occur again. The simplest version of this scheme is shown in Fig. 4-5(a). Here the history table contains one entry for each conditional branch instruction. The entry contains the address of the branch instruction along with a bit telling whether it was taken the last time it was executed. Using this scheme, the prediction is simply that the branch will go the same way it went last time. If the prediction is wrong, the bit in the history table is changed.

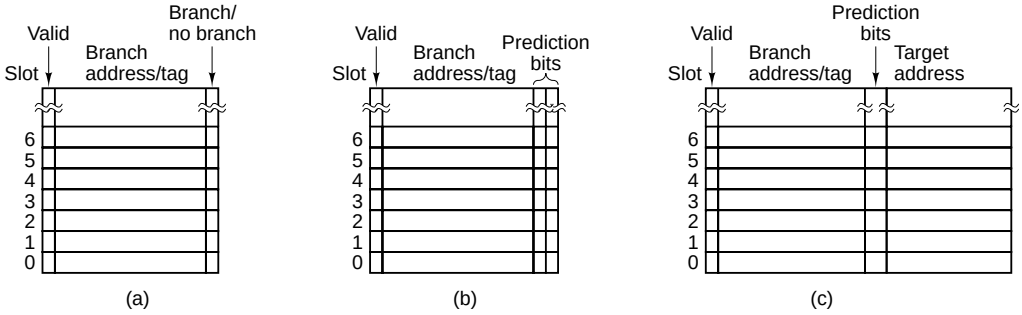


Figure 4-5. (a) A 1-bit branch history. (b) A 2-bit branch history. (c) A mapping between branch instruction address and target address.

There are several ways to organize the history table. In fact, these are precisely the same ways used to organize a cache. Consider a machine with 32-bit instructions that are word aligned so that the low-order 2 bits of each memory address are 00. With a direct-mapped history table containing 2^n entries, the low-order $n + 2$ bits of a branch instruction can be extracted and shifted right 2 bits. This n-bit number can be used as an index into the history table where a check is made to see if the address stored there matches the address of the branch. As with a cache, there is no need to store the low-order $n + 2$ bits, so they can be omitted (i.e., just the upper address bits—the tag—are stored). If there is a hit, the prediction bit is used to predict the branch. If the wrong tag is present or the entry is invalid, a miss occurs, just as with a cache. In this case, the forward/backward branch rule can be used.

If the branch history table has, say, 4096 entries, then branches at addresses 0, 16384, 32768, ... will conflict, analogous to the same problem with a cache. The same solution is possible: a two-way, four-way, or n-way associative entry. As with a cache, the limiting case is a single n-way associative entry, which requires full associativity of lookup.

Given a large enough table size and enough associativity, this scheme works well in most situations. However, one systematic problem always occurs. When a loop is finally exited, the branch at the end will be mispredicted, and worse yet, the misprediction will change the bit in the history table to indicate a future prediction of ‘‘no branch.’’ The next time the loop is entered, the branch at the end of

the first iteration will be predicted wrong. If the loop is inside an outer loop, or in a frequently-called procedure, this error can occur often.

To eliminate this misprediction, we can give the table entry a second chance. With this method, the prediction is only changed after two consecutive incorrect predictions. This approach requires having two prediction bits in the history table, one for what the branch is ''supposed'' to do, and one for what it did last time, as shown in Fig. 4-5(b).

A slightly different way of looking at this algorithm is to see it as a finite-state machine with four states, as depicted in Fig. 4-6. After a series of consecutive successful ''no branch'' predictions, the FSM will be in state 00 and will predict ''no branch'' next time. If that prediction is wrong, it will move to state 10, but predict ''no branch'' next time as well. Only if this prediction is wrong will it now move to state 11 and predict branches all the time. In effect, the leftmost bit of the state is the prediction and the rightmost bit is what the branch did last time. While this design uses only 2 bits of history, a design that keeps track of 4 or 8 bits of history is also possible.

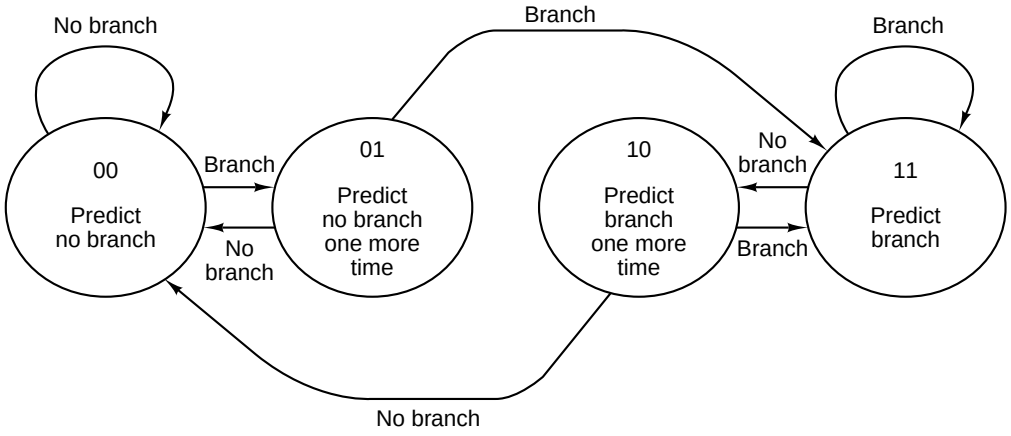


Figure 4-6. A 2-bit finite-state machine for branch prediction.

This is not our first FSM. Fig. 4-0 was also an FSM. In fact, all of our microprograms can be regarded as FSMs, since each line represents a specific state the machine can be in, with well-defined transitions to a finite set of other states. FSMs are very widely used in all aspects of hardware design.

So far, we have assumed that the target of each conditional branch was known, typically either as an explicit address to branch to (contained within the instruction itself), or as a relative offset from the current instruction (i.e., a signed number to add to the program counter). Often this assumption is valid, but some conditional branch instructions compute the target address by doing arithmetic on registers, and then going there. Even if the FSM of Fig. 4-6 accurately predicts the branch will be taken, such a prediction is of no use if the target address is unknown. One way of dealing with this situation is to store the actual address

branched to last time in the history table, as shown in Fig. 4-5(c). In this way, if the table says that the last time the branch at address 516 was taken it went to address 4000, if the prediction is now for ''branch,'' the working assumption will be a branch to 4000 again.

A different approach to branch prediction is to keep track of whether the last *k* conditional branches encountered were taken, irrespective of which instructions they were (Pan et al., 1992). This *k*-bit number, kept in the **branch history shift register**, is then compared in parallel to all the entries of a history table with a *k*-bit key and if a hit occurs, the prediction found there used. Somewhat surprisingly, this technique works quite well.

Static Branch Prediction

All of the branch prediction techniques discussed so far are dynamic, that is, carried out at run time while the program is running. They also adapt to the program's current behavior, which is good. The down side is that they require specialized and expensive hardware and a great deal of chip complexity.

A different way to go is to have the compiler help out. When the compiler sees a statement like

```
for (i = 0; i < 1000000; i++) { ... }
```

it knows very well that the branch at the end of the loop will be taken nearly all the time. If only there were a way for it to tell the hardware, a lot of effort could be saved.

Although this is an architectural change (and not just an implementation issue), some machines, such as the UltraSPARC II, have a second set of conditional branch instructions, in addition to the regular ones (which are needed for backward compatibility). The new ones contain a bit in which the compiler can specify that it thinks the branch will be taken (or not taken). When one of these is encountered, the fetch unit just does what it has been told. Furthermore, there is no need to waste precious space in the branch history table for these instructions, thus reducing conflicts there.

Finally, our last branch prediction technique is based on profiling (Fisher and Freudenberger, 1992). This, too, is a static technique, but instead of having the compiler try to figure out which branches will be taken and which will not, the program is actually run (typically on a simulator), and the branch behavior captured. This information is fed into the compiler, which then uses the special conditional branch instructions to tell the hardware what to do.

4.5.3 Out-of-Order Execution and Register Renaming

Most modern CPUs are both pipelined and superscalar, as shown in Fig. 2-6. What this generally means is that there is a fetch unit that pulls instruction words out of memory before they are needed in order to feed a decode unit. The decode unit issues the decoded instructions to the proper functional units for execution. In some cases it may break individual instructions into micro-ops before issuing them to the functional units, depending on what the functional units are capable of doing.

Clearly, the machine design is simplest if all instructions are executed in the order they are fetched (assuming for the moment that the branch prediction algorithm never guesses wrong). However, in-order execution does not always give optimal performance due to dependences between instructions. If an instruction needs a value computed by the previous instruction, the second one cannot begin executing until the first one has produced the needed value. In this situation (a RAW dependence), the second instruction has to wait. Other kinds of dependences also exist, as we will soon see.

In an attempt to get around these problems and produce better performance, some CPUs allow dependent instructions to be skipped over, to get to future instructions that are not dependent. Needless to say, the internal instruction scheduling algorithm used must deliver the same effect as if the program were executed in the order written. We will now demonstrate how instruction reordering works using a detailed example.

To illustrate the nature of the problem, we will start with a machine that always issues instructions in program order and also requires them to complete execution in program order. The significance of the latter point will become clear later.

Our example machine has eight registers visible to the programmer, `R0` through `R7`. All arithmetic instructions use three registers: two for the operands and one for the result, the same as the Mic-4. We will assume that if an instruction is decoded in cycle n, execution starts in cycle $n + 1$. For a simple instruction, such as an addition or subtraction, the writeback to the destination register occurs at the end of cycle $n + 2$. For a more complicated instruction, such as a multiplication, the writeback occurs at the end of cycle $n + 3$. To make the example realistic, we will allow the decode unit to issue up to two instructions per clock cycle. Commercial superscalar CPUs often can issue four or even six instructions per clock cycle.

Our example execution sequence is shown in Fig. 4-7. Here the first column gives the number of the cycle and the second one gives the instruction number. The third column lists the instruction decoded. The fourth one tells which instruction is being issued (with a maximum of two per clock cycle). The fifth one tells which instruction has been retired (completed). Remember that in this example we are requiring both in-order issue and in-order completion, so instruction $k + 1$

cannot be issued until instruction *k* has been issued, and instruction *k* + 1 cannot be retired (meaning the writeback to the destination register is performed) until instruction *k* has been retired. The other 16 columns are discussed below.

					Registers being read								Registers being written							
Cy	**#**	**Decoded**	**Iss**	**Ret**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	–		3	2	1									1	1	1		
3					3	2	1									1	1	1		
4				1	2	1	1										1	1		
				2	1	1												1		
				3																
5			4			1			1										1	
	5	R7=R1*R2	5			2	1		1										1	1
6	6	R1=R0–R2	–			2	1		1										1	1
7				4		1	1													1
8				5																
9			6		1		1							1						
	7	R3=R3*R1	7		1	1	1	1						1		1				
10					1	1	1	1						1		1				
11				6		1		1								1				
12				7																
13	8	R1=R4+R4	8						2					1						
14									2					1						
15				8																

Figure 4-7. Operation of a superscalar CPU with in-order issue and in-order completion.

After decoding an instruction, the decode unit has to decide whether or not it can be issued immediately. To make this decision, the decode unit needs to know the status of all the registers. If, for example, the current instruction needs a register whose value has not yet been computed, the current instruction cannot be issued and the CPU must stall.

We will keep track of register use with a device called a **scoreboard**, which was first present in the CDC 6600. The scoreboard has a small counter for each register telling how many times that register is in use as a source by currently-executing instructions. If a maximum of, say, 15 instructions may be executing at once, then a 4-bit counter will do. When an instruction is issued, the scoreboard entries for its operand registers are incremented. When an instruction is retired,

the entries are decremented.

The scoreboard also has a counter for each register to keep track of registers being used as destinations. Since only one write at a time is allowed, these counters can be 1-bit wide. The rightmost 16 columns in Fig. 4-7 show the scoreboard.

In real machines, the scoreboard also keeps track of functional unit usage, to avoid issuing an instruction for which no functional unit is available. For simplicity, we will assume there is always a suitable functional unit available, so we will not show the functional units on the scoreboard.

The first line of Fig. 4-7 shows I1 (instruction 1), which multiplies R0 by R1 and puts the result in R3. Since none of these registers are in use yet, the instruction is issued and the scoreboard is updated to reflect that R0 and R1 are being read and R3 is being written. No subsequent instruction can write into any of these or can read R3 until I1 has been retired. Since this instruction is a multiplication, it will be finished at the end of cycle 4. The scoreboard values shown on each line reflect their state after the instruction on that line has been issued. Blank entries are 0s.

Since our example is a superscalar machine that can issue two instructions per cycle, a second instruction (I2) is issued during cycle 1. It adds R0 and R2, storing the result in R4. To see if this instruction can be issued, the following rules are applied:

1. If any operand is being written, do not issue (RAW dependence).
2. If the result register is being read, do not issue (WAR dependence).
3. If the result register is being written, do not issue (WAW dependence).

We have already seen RAW dependences, which occur when an instruction needs to use as a source a result that a previous instruction has not yet produced. The other two dependences are less serious. They are essentially resource conflicts. In a **WAR dependence** (Write After Read), one instruction is trying to overwrite a register that a previous instruction may not yet have finished reading. A **WAW dependence** (Write After Write) is similar. These can often be avoided by having the second instruction put its results somewhere else (perhaps temporarily). If none of the above three dependences exist, and the functional unit it needs is available, the instruction is issued. In this case, I2 uses a register (R0) that is being read by a pending instruction, but this overlap is permitted so I2 is issued. Similarly, I3 is issued during cycle 2.

Now we come to I4, which needs to use R4. Unfortunately, we see from line 3 that R4 is being written. Here we have a RAW dependence, so the decode unit stalls until R4 becomes available. While stalled, it stops pulling instructions from the fetch unit. When the fetch unit's internal buffers fill up, it will stop prefetching.

It is worth noting that the next instruction in program order, I5, does not have

conflicts with any of the pending instructions. It could have been decoded and issued were it not for the fact that this design requires issuing instructions in order.

Now let us look at what happens during cycle 3. I2, being an addition (two cycles), finishes at the end of cycle 3. Unfortunately, it cannot be retired (thus freeing up R4 for I4). Why not? The reason is that this design also requires in-order retirement. Why? What harm could possibly come from doing the store into R4 now and marking it as available?

The answer is subtle, but important. Suppose that instructions could complete out of order. Then if an interrupt occurred, it would be very difficult to save the state of the machine so it could be restored later. In particular, it would not be possible to say that all instructions up to some address had been executed and all instructions beyond it had not. This is called a **precise interrupt** and is a desirable characteristic in a CPU (Moudgill and Vassiliadis, 1996). Out-of-order retirement makes interrupts imprecise, which is why some machines require in-order instruction completion.

Getting back to our example, at the end of cycle 4, all three pending instructions can be retired, so in cycle 5 I4 can finally be issued, along with the newly decoded I5. Whenever an instruction is retired, the decode unit has to check to see if there is a stalled instruction that can now be issued.

In cycle 6, I6 stalls because it needs to write into R1 and R1 is busy. It is finally started in cycle 9. The entire sequence of eight instructions takes 15 cycles to complete due to many dependences, even though the hardware is capable of issuing two instructions on every cycle. Notice, however, that when reading down the *Iss* column of Fig. 4-7, all the instructions have been issued in order. Likewise, the *Ret* column shows that they have been retired in order as well.

Now let us consider an alternative design: out-of-order execution. In this design, instructions may be issued out of order and may be retired out of order as well. The same sequence of eight instructions is shown in Fig. 4-8, only now with out-of-order issue and out-of-order retirement permitted.

The first difference occurs in cycle 3. Even though I4 has stalled, we can decode and issue I5 since it does not conflict with any pending instruction. However, skipping over instructions causes a new problem. Suppose that I5 had used an operand computed by the skipped instruction, I4. With the current scoreboard, we would not have noticed this. As a consequence, we have to extend the scoreboard to keep track of stores done by skipped-over instructions. This can be done by adding a second bit map, 1 bit per register, to keep track of stores done by stalled instructions. (These counters are not shown in the figure.) The rule for issuing instructions now has to be extended to prevent the issue of any instruction with an operand scheduled to be stored into by an instruction that came before it but was skipped over.

Now let us look back at I6, I7, and I8 in Fig. 4-7. Here we see that I6 computes a value in R1 that is used by I7. However, we also see that the value is

					Registers being read								Registers being written							
Cy	#	Decoded	Iss	Ret	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	–		3	2	1									1	1	1		
3	5	R7=R1*R2	5		3	3	2									1	1	1		1
	6	S1=R0–R2	6		4	3	3									1	1	1		1
				2	3	3	2									1		1		1
4			4		3	4	2		1							1		1	1	1
	7	R3=R3*S1	–		3	4	2		1							1		1	1	1
	8	S2=R4+R4	8		3	4	2		3							1		1	1	1
				1	2	3	2		3									1	1	1
				3	1	2	2		3										1	1
5				6		2	1		3					1					1	1
6			7			2	1	1	3					1		1			1	1
				4		1	1	1	2					1		1				1
				5				1	2					1		1				
				8				1								1				
7								1								1				
8								1								1				
9				7																

Figure 4-8. Operation of a superscalar CPU with out-of-order issue and out-of-order completion.

never used again because I8 overwrites R1. There is no real reason to use R1 as the place to hold the result of I6. Worse yet, R1 is a terrible choice of intermediate register, although a perfectly reasonable one for a compiler or programmer used to the idea of sequential execution with no instruction overlap.

In Fig. 4-8 we introduce a new technique for solving this problem: **register renaming**. The wise decode unit changes the use of R1 in I6 (cycle 3) and I7 (cycle 4) to a secret register, S1, not visible to the programmer. Now I6 can be issued concurrently with I5. Modern CPU often have dozens of secret registers for use with register renaming. This technique can often eliminate WAR and WAW dependences.

At I8, we use register renaming again. This time R1 is renamed into S2 so the addition can be started before R1 is free, at the end of cycle 6. If it turns out that the result really has to be in R1 this time, the contents of S2 can always be copied back there just in time. Even better, all future instructions needing it can have their sources renamed to the register where it really is stored. In any case, the I8 addition got to start earlier this way.

On many real machines, renaming is deeply embedded in the way the registers are organized. There are many secret registers and a table that maps the registers visible to the programmer onto the secret registers. Thus the real register being used for, say, R0 is located by looking at entry 0 of this mapping table. In this way, there is no real register R0, just a binding between the name R0 and one of the secret registers. This binding changes frequently during execution to avoid dependences.

Notice that in Fig. 4-8 when reading down the fourth column, the instructions have not been issued in order. Nor they have been retired in order. The conclusion of this example is simple: using out-of-order execution and register renaming we were able to speed up the computation by close to a factor of two.

4.5.4 Speculative Execution

In the previous section we introduced the concept of reordering instructions in order to improve performance. Although we did not mention it explicitly, the focus there was on reordering instructions within a single basic block. It is now time to look at this point more closely.

Computer programs can be broken up into **basic blocks**, with each basic block consisting of a linear sequence of code with one entry point on top and one exit on the bottom. A basic block does not contain any control structures (e.g., if statements or while statements) so that its translation into machine language does not contain any branches. The basic blocks are connected by control statements.

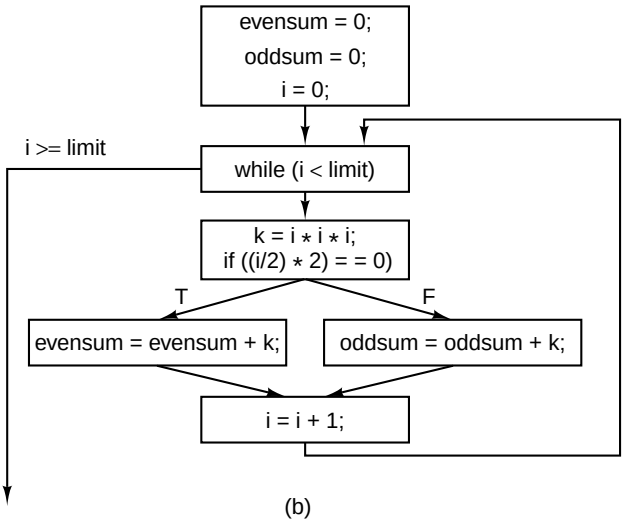
A program in this form can be represented as a directed graph, as shown in Fig. 4-9. Here we compute the sum of the cubes of the even and odd integers up to some limit and accumulate them in *evensum* and *oddsum*, respectively. Within each basic block, the reordering techniques of the previous section work fine.

The trouble is that most basic blocks are short and there is insufficient parallelism in them to exploit effectively. Consequently, the next step is to allow the reordering to cross basic block boundaries in an attempt to fill all the issue slots. The biggest gains come when a potentially slow operation can be moved upward in the graph to start it going early. This might be a LOAD instruction, a floating-point operation, or even the start of a long dependence chain. Moving code upward over a branch is called **hoisting**.

Imagine that in Fig. 4-9 all the variables were kept in registers except *evensum* and *oddsum* (for lack of registers). It might make sense then to move their LOAD instructions to the top of the loop, before computing *k*, to get them started early on, so the values will available when they are needed. Of course, only one of them will be needed on each iteration, so the other LOAD will be wasted, but if the cache and memory are pipelined and there are issue slots available, it might still be worth doing this. Executing code before it is known if it is even going to be needed is called **speculative execution**. Using this technique requires support from the compiler and the hardware as well as some architectural extensions. In

```
evensum = 0;
oddsum = 0;
i = 0;
while (i < limit) {
    k = i * i * i;
    if ((i/2) * 2) == 0)
        evensum = evensum + k;
    else
        oddsum = oddsum + k;
    i = i + 1;
}
```

(a)



(b)

Figure 4-9. (a) A program fragment. (b) The corresponding basic block graph.

most cases, reordering instructions over basic block boundaries is beyond the capability of the hardware, so the compiler must move the instructions explicitly.

Speculative execution introduces some interesting problems. For one, it is essential that none of the speculative instructions have irrevocable results because it may turn out later that they should not have been executed. In Fig. 4-9, it is fine to fetch *evensum* and *oddsum*, and it is also fine to do the addition as soon as *k* is available (even before the if statement), but it is not fine to store the results back in memory. In more complicated code sequences, one common way of preventing speculative code from overwriting registers before it is known if this is desired is to rename all the destination registers used by the speculative code. In this way, only scratch registers are modified, so there is no problem if the code ultimately is not needed. If the code is needed, the scratch registers are copied to the true destination registers. As you can imagine, the scoreboarding to keep track of all this is not simple, but given enough hardware, it can be done.

However, there is another problem introduced by speculative code that cannot be solved by register renaming. What happens if a speculatively executed instruction causes an exception? A painful, but not fatal, example is a LOAD instruction that causes a cache miss on a machine with a large cache line size (say, 256 bytes) and a memory far slower than the CPU and cache. If a LOAD that is actually needed stops the machine dead in its tracks for many cycles while the cache line is being loaded, well, that's life, since the word is needed. However, stalling the machine to fetch a word that turns out not to be needed is counterproductive. Too many of these ''optimizations,'' may make the CPU slower than if it did not have them at all. (If the machine has virtual memory, which is discussed in Chap. 6, a

speculative LOAD might even cause a page fault, which requires a disk operation to bring in the needed page. False page faults can have a terrible effect on performance, so it is important to avoid them.)

One solution present in a number of modern machines is to have a special SPECULATIVE-LOAD instruction that tries to fetch the word from the cache, but if it is not there, just gives up. If the value is there when it is actually needed, it can be used, but if it is not, the hardware must go out and get it on the spot. If the value turns out not to be needed, no penalty has been paid for the cache miss.

A far worse situation can be illustrated with the following statement:

```
if (x > 0) z = y/x;
```

where *x*, *y*, and *z* are floating-point variables. Suppose that the variables are all fetched into registers in advance and that the (slow) floating-point division is hoisted above the if test. Unfortunately, *x* is 0 and the resulting divide-by-zero trap terminates the program. The net result is that speculation has caused a correct program to fail. Worse yet, the programmer put in explicit code to prevent this situation and it happened anyway. This situation is not likely to lead to a happy programmer.

One possible solution is to have special versions of instructions that might cause exceptions. In addition, a bit, called a **poison bit**, is added to each register. When a special speculative instruction fails, instead of causing a trap, it sets the poison bit on the result register. If that register is later touched by a regular instruction, the trap occurs then (as it should). However, if the result is never used, the poison bit is eventually cleared and no harm is done.

5

THE INSTRUCTION SET ARCHITECTURE LEVEL

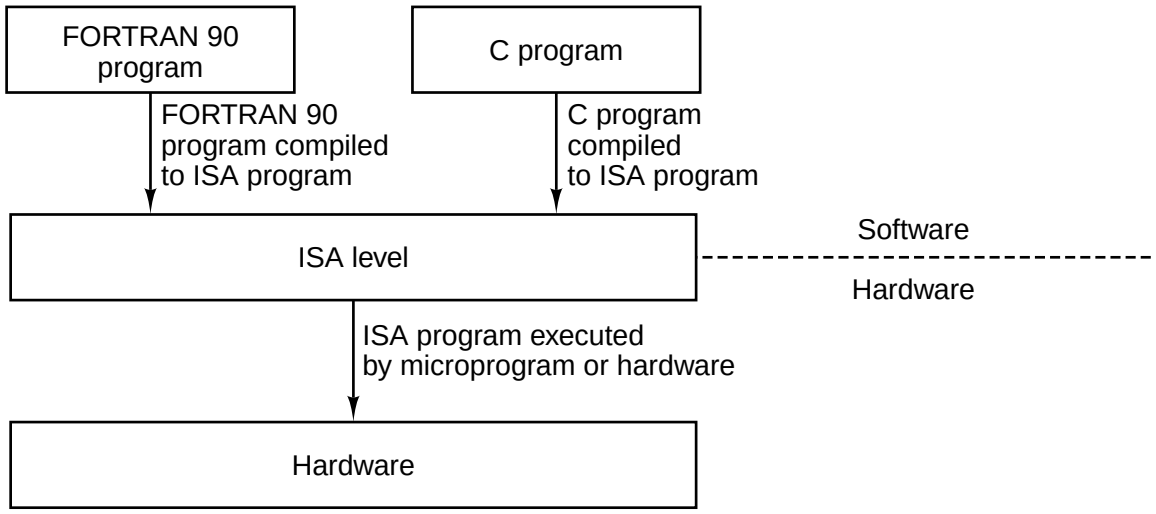


Figure 5-1. The ISA level is the interface between the compilers and the hardware.

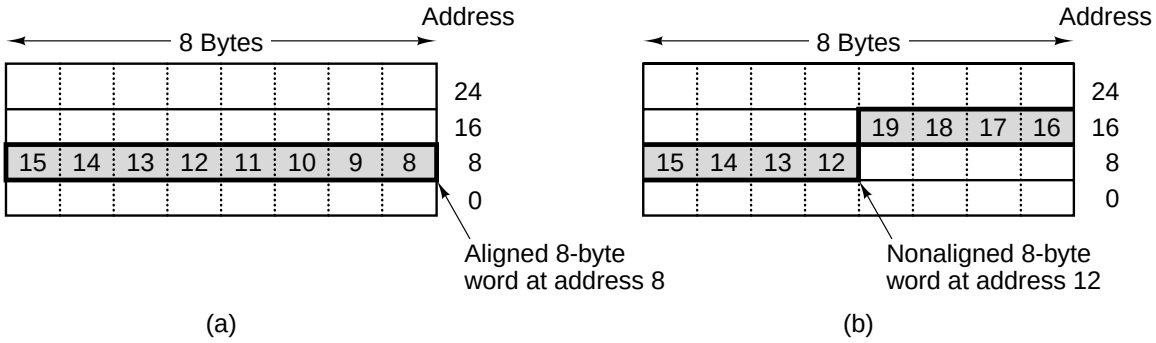


Figure 5-2. An 8-byte word in a little-endian memory. (a) Aligned. (b) Not aligned. Some machines require that words in memory be aligned.

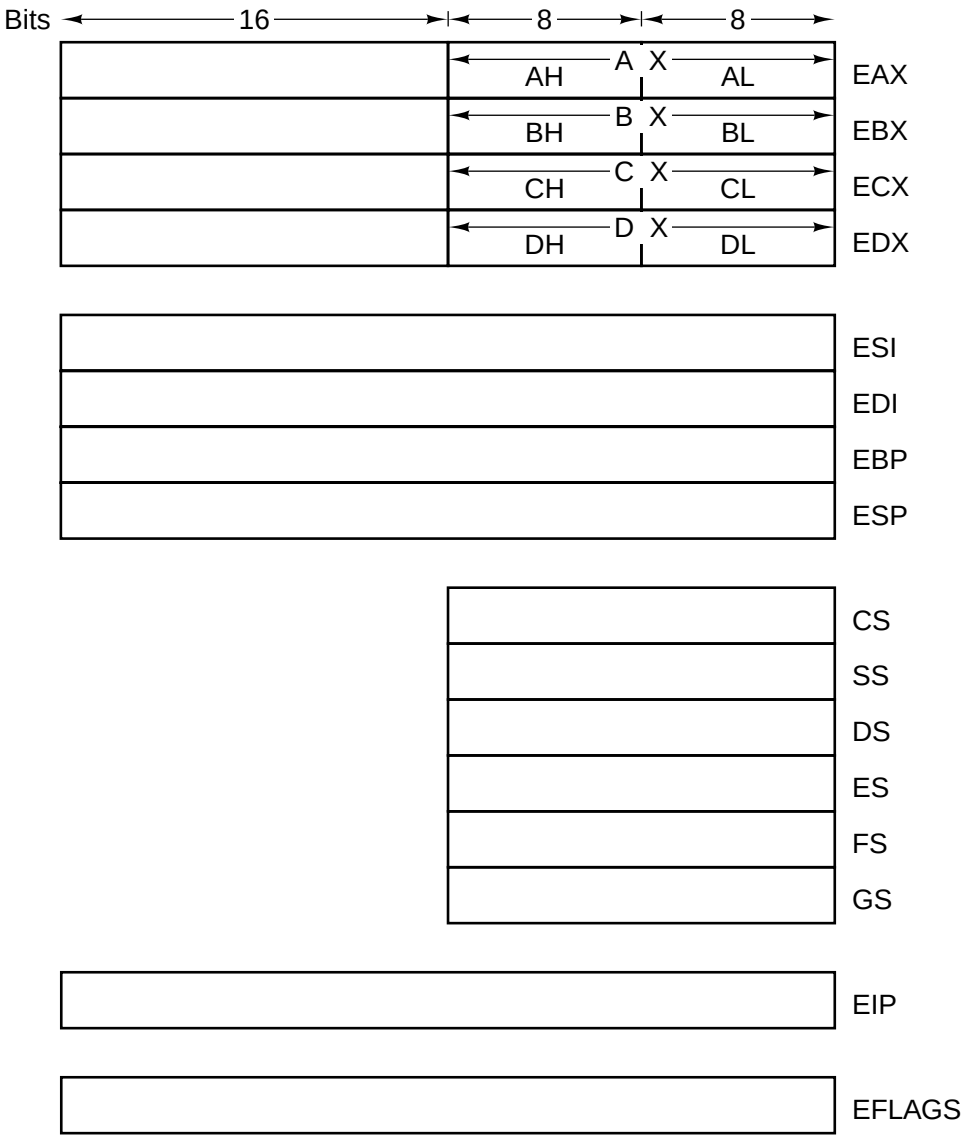


Figure 5-3. The Pentium II's primary registers.

Register	Alt. name	Function
R0	G0	Hardwired to 0. Stores into it are just ignored.
R1 – R7	G1 – G7	Holds global variables
R8 – R13	O0 – O5	Holds parameters to the procedure being called
R14	SP	Stack pointer
R15	O7	Scratch register
R16 – R23	L0 – L7	Holds local variables for the current procedure
R24 – R29	I0 – I5	Holds incoming parameters
R30	FP	Pointer to the base of the current stack frame
R31	I7	Holds return address for the current procedure

Figure 5-4. The UltraSPARC II's general registers.

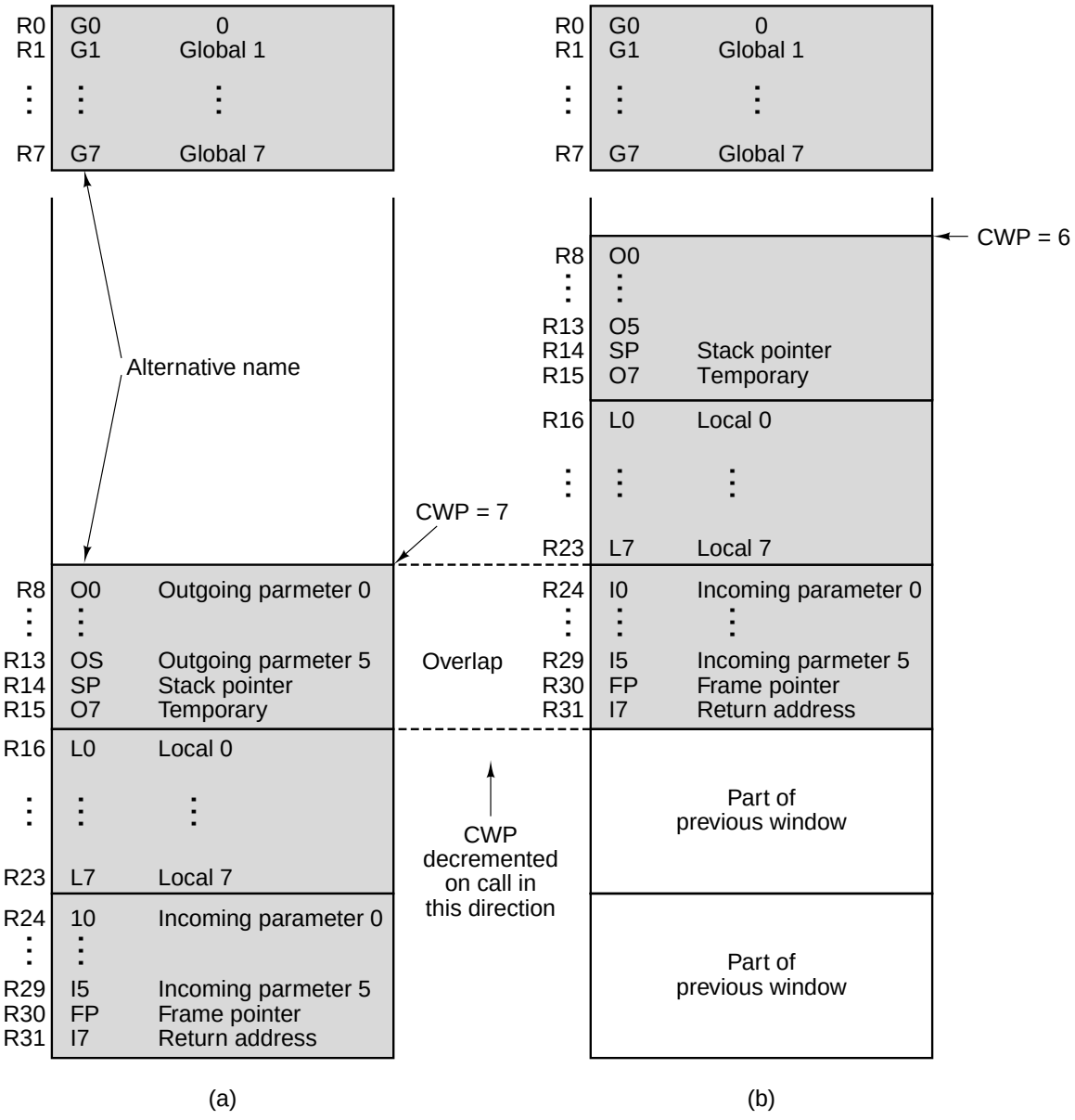


Figure 5-5. Operation of the UltraSPARC II register windows.

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×		
Unsigned integer	×	×	×		
Binary coded decimal integer	×				
Floating point			×	×	

Figure 5-6. The Pentium II numeric data types. Supported types are marked with ×.

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×	×	
Unsigned integer	×	×	×	×	
Binary coded decimal integer					
Floating point			×	×	×

Figure 5-7. The UltraSPARC II numeric data types.

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×	×	
Unsigned integer					
Binary coded decimal integer					
Floating point			×	×	

Figure 5-8. The JVM numeric data types.

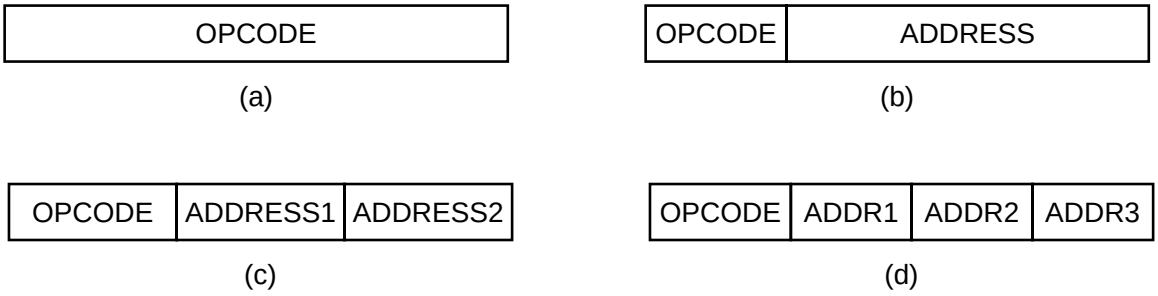


Figure 5-9. Four common instruction formats: (a) Zero-address instruction. (b) One-address instruction (c) Two-address instruction. (d) Three-address instruction.

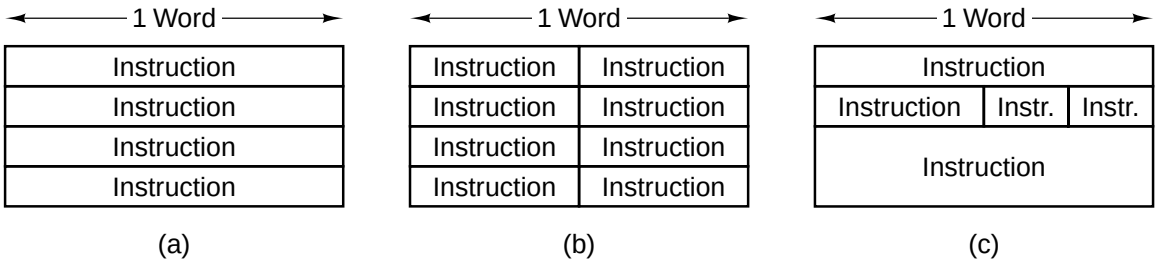


Figure 5-10. Some possible relationships between instruction and word length.

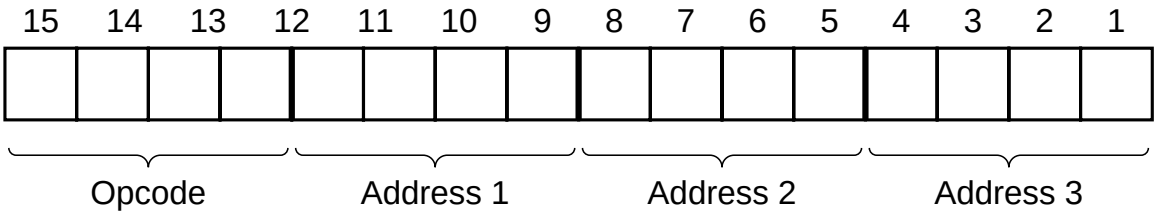


Figure 5-11. An instruction with a 4-bit opcode and three 4-bit address fields.

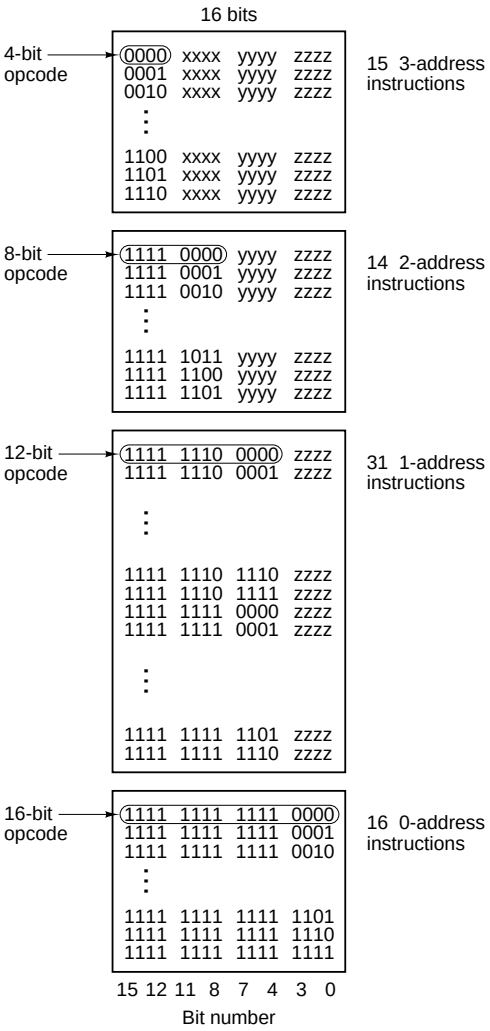


Figure 5-12. An expanding opcode allowing 15 three-address instructions, 14 two-address instructions, 31 one-address instructions, and 16 zero-address instructions. The fields marked *xxxx*, *yyyy*, and *zzzz* are 4-bit address fields.

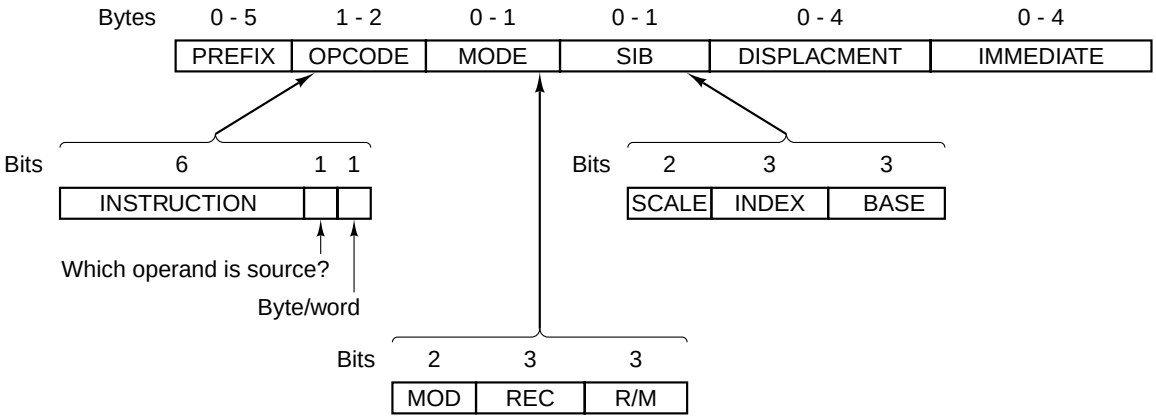


Figure 5-13. The Pentium II instruction formats.

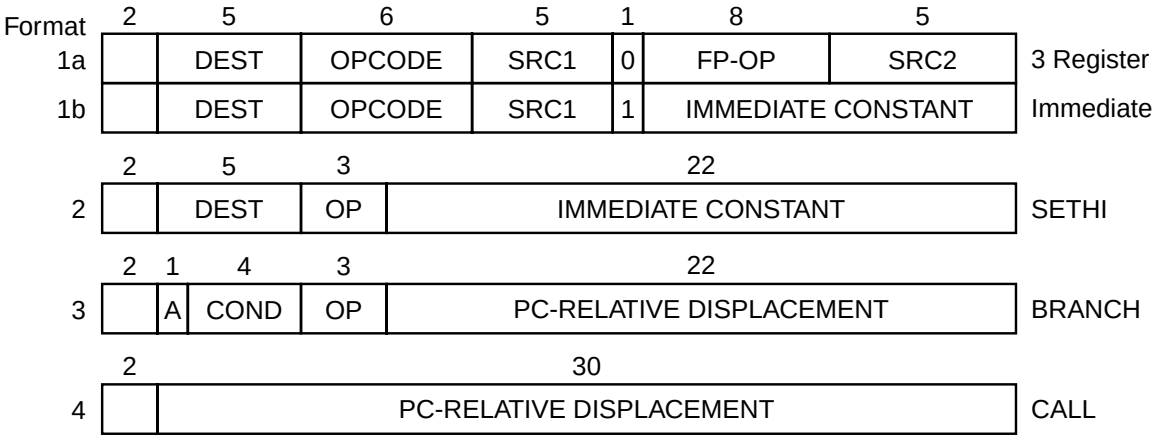


Figure 5-14. The original SPARC instruction formats.

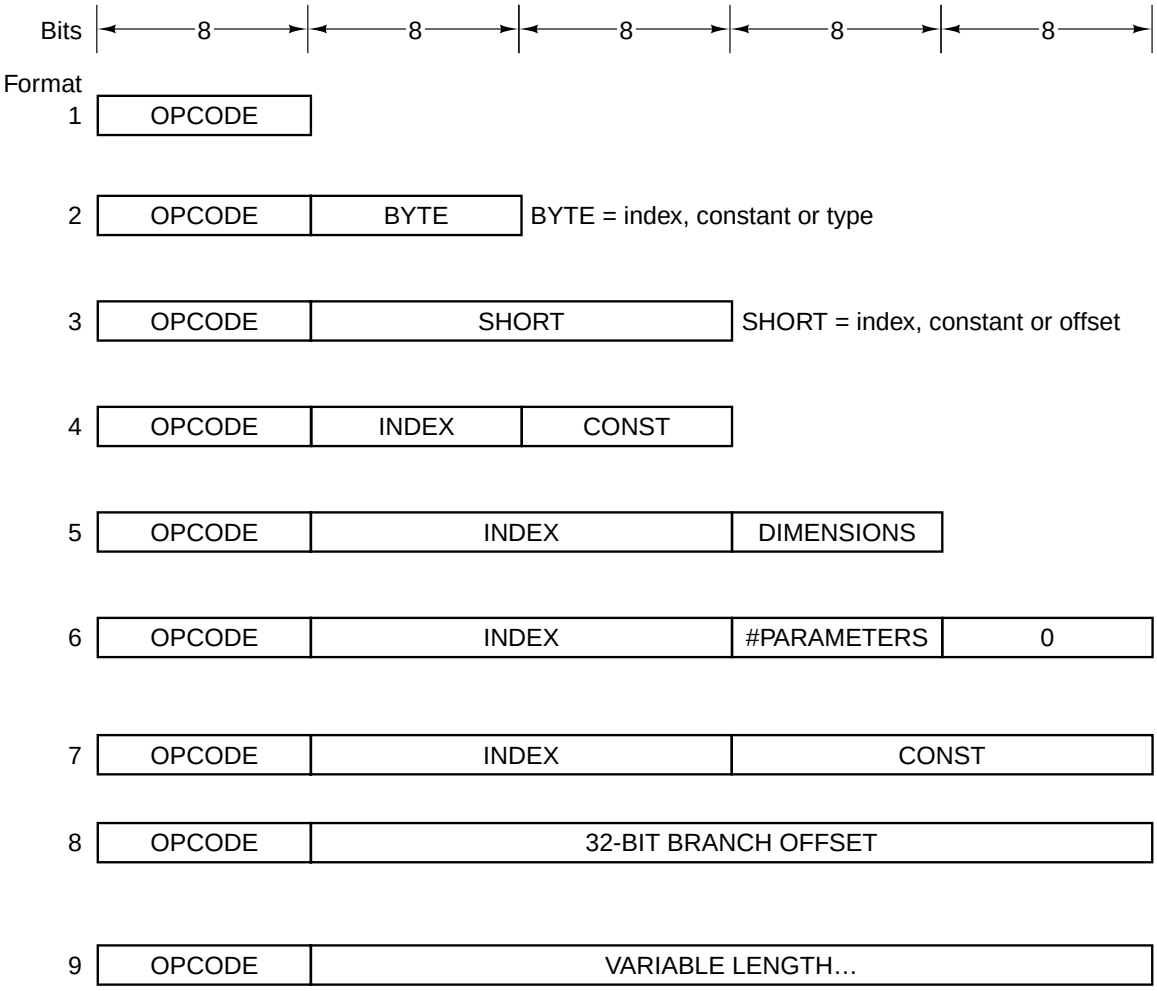


Figure 5-15. The JVM instruction formats.

MOV	R1	4

Figure 5-16. An immediate instruction for loading 4 into register 1.

```
        MOV R1,#0     ; accumulate the sum in R1, initially 0
        MOV R2,#A     ; R2 = address of the array A
        MOV R3,#A+1024; R3 = address if the first word beyond A
LOOP:               ADD R1,(R2); register indirect through R2 to get operand
        ADD R2,#4     ; increment R2 by one word (4 bytes)
        CMP R2,R3     ; are we done yet?
        BLT LOOP      ; if R2 < R3, we are not done, so continue
```

Figure 5-17. A generic assembly program for computing the sum of the elements of an array.

```
        MOV R1,#0      ; accumulate the OR in R1, initially 0
        MOV R2,#0      ; R2 = index, i, of current product: A[i] AND B[i]
        MOV R3,#4096; R3 = first index value not to use
LOOP:                  MOV R4,A(R2); R4 = A[i]
        AND R4,B(R2); R4 = A[i] AND B[i]
        OR R1,R4       ; OR all the Boolean products into R1
        ADD R2,#4      ; i = i + 4 (step in units of 1 word = 4 bytes)
        CMP R2,R3      ; are we done yet?
        BLT LOOP       ; if R2 < R3, we are not done, so continue
```

Figure 5-18. A generic assembly program for computing the OR of A_i AND B_i for two 1024-element arrays.

MOV	R4	R2	124300

Figure 5-19. A possible representation of MOV R4,A(R2).

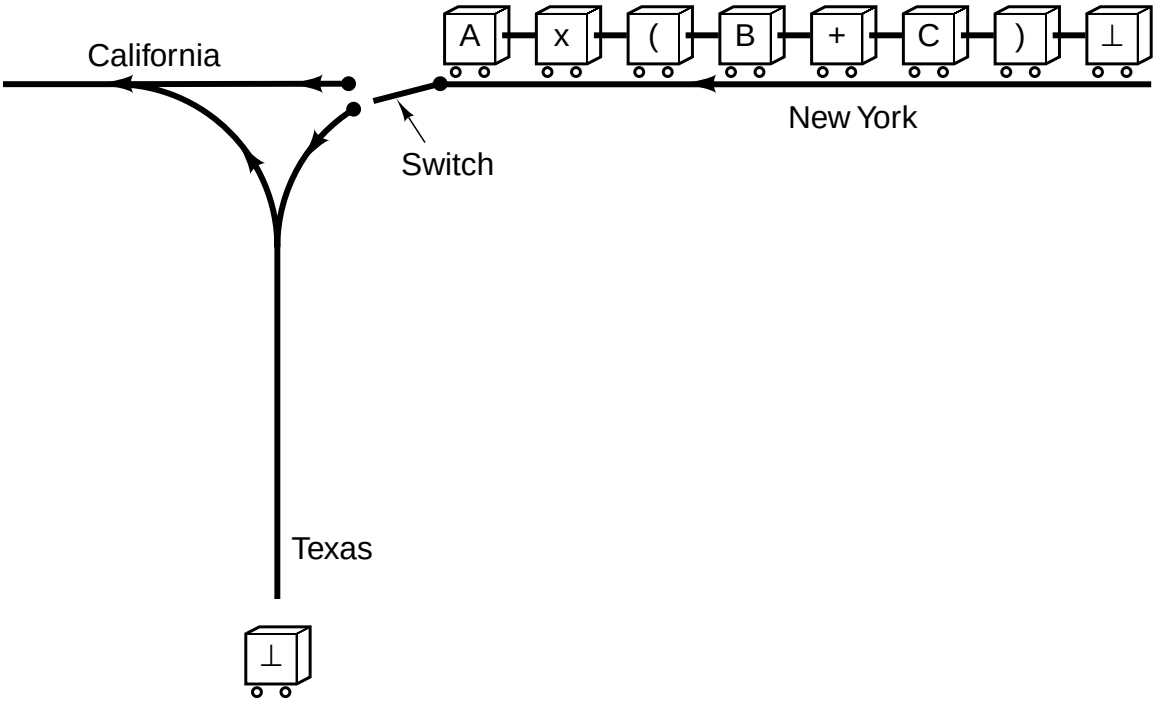


Figure 5-20. Each railroad car represents one symbol in the formula to be converted from infix to reverse Polish notation.

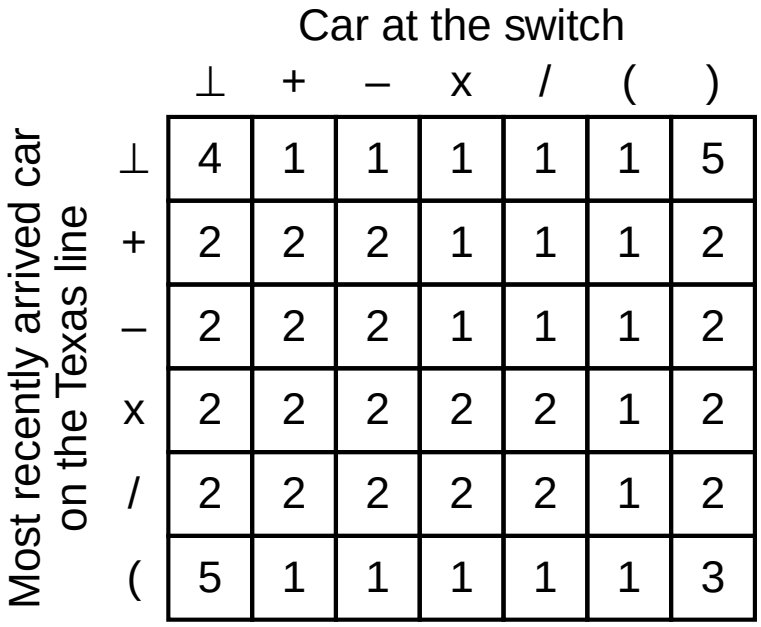


Figure 5-21. Decision table used by the infix-to-reverse Polish notation algorithm

Infix	Reverse Polish notation
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C – D)	A B + C D – /
A × B / C	A B × C /
((A + B) × C + D)/(E + F + G)	A B + C × D + E F + G + /

Figure 5-22. Some examples of infix expressions and their reverse Polish notation equivalents.

Step	Remaining string	Instruction	Stack
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	BIPUSH 2	18, 1, 3, 2
9	× + 4 – /	IMUL	18, 1, 6
10	+ 4 – /	IADD	18, 7
11	4 – /	BIPUSH 4	18, 7, 4
12	– /	ISUB	18, 3
13	/	IDIV	6

Figure 5-23. Use of a stack to evaluate a reverse Polish notation formula.

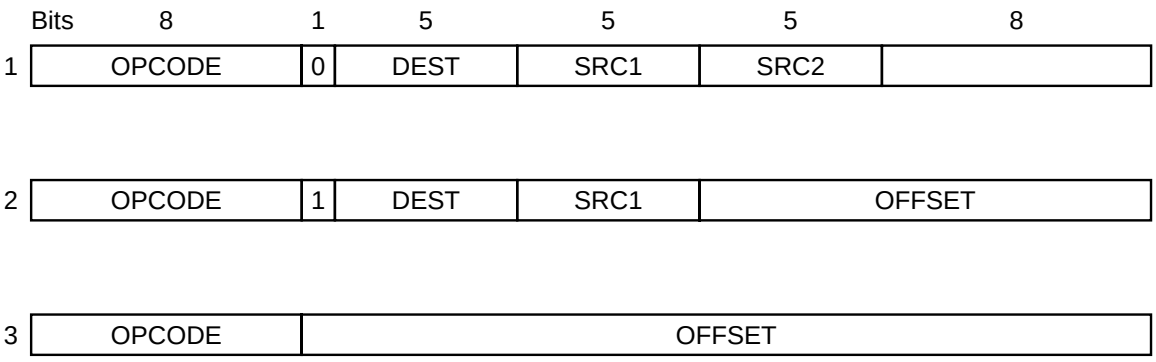


Figure 5-24. A simple design for the instruction formats of a three-address machine.

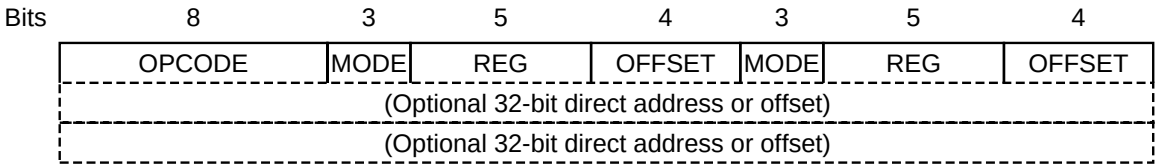


Figure 5-25. A simple design for the instruction formats of a two-address machine.

	MOD			
R/M	**00**	**01**	**10**	**11**
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX or AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX or CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX or DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX or BL
100	SIB	SIB with OFFSET8	SIB with OFFSET32	ESP or AH
101	Direct	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP or CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI or DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI or BH

Figure 5-26. The Pentium II 32-bit addressing modes. M[*x*] is the memory word at *x*.

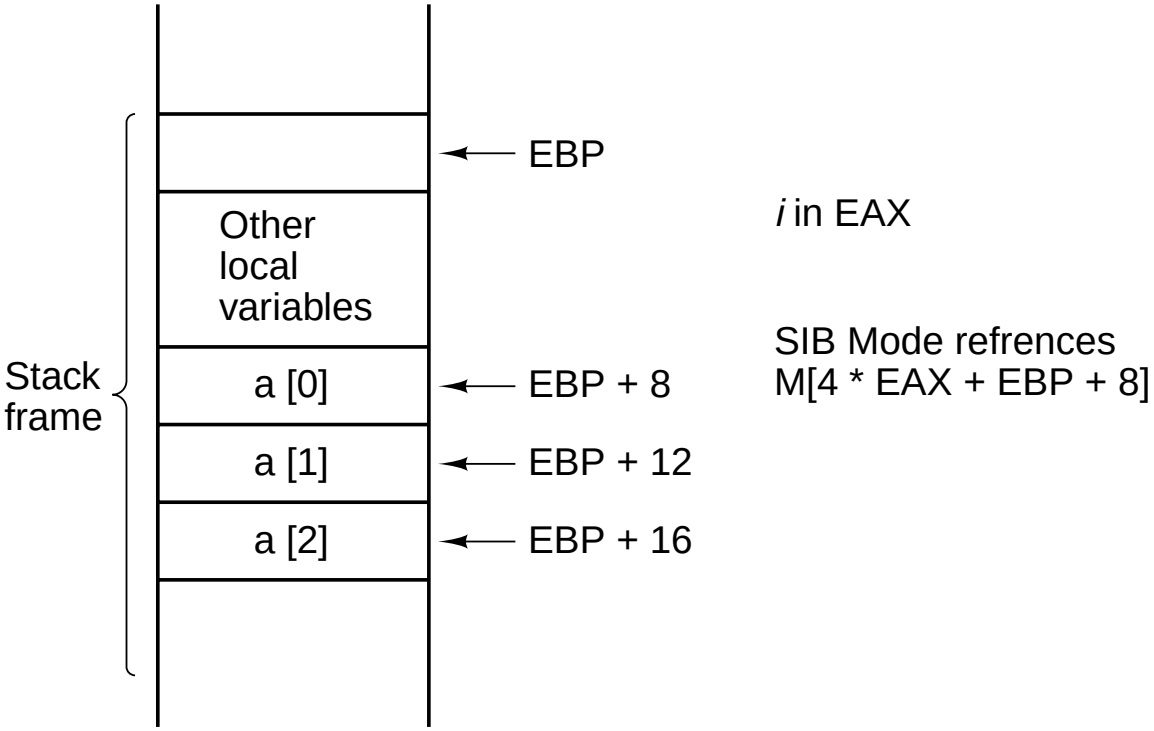


Figure 5-27. Access to *a*[*i*].

Addressing mode	Pentium II	UltraSPARC II	JVM
Immediate	×	×	×
Direct	×		
Register	×	×	
Register indirect	×		
Indexed	×	×	×
Based-indexed		×	
Stack			×

Figure 5-28. A comparison of addressing modes.

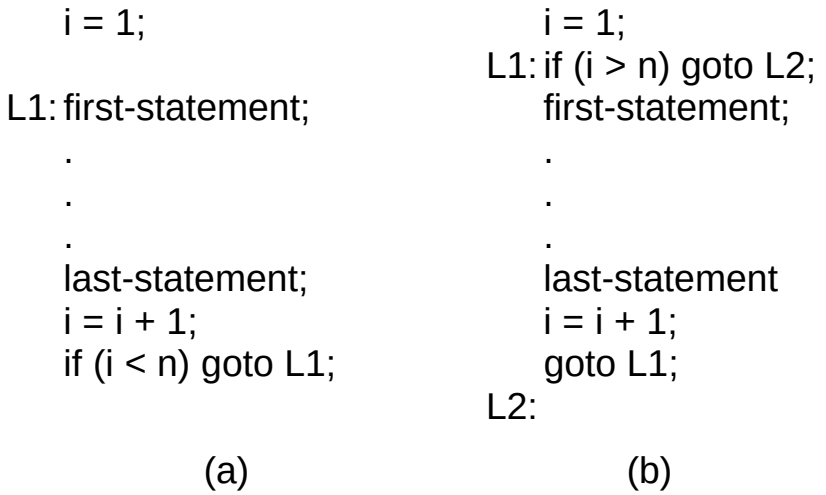


Figure 5-29. (a) Test-at-the-end loop. (b) Test-at-the-beginning loop.

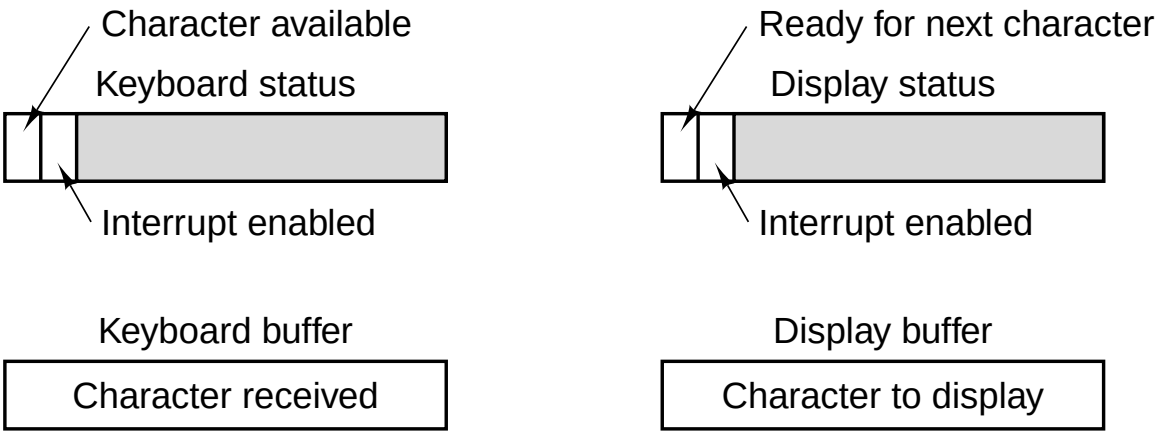


Figure 5-30. Device registers for a simple terminal.

```
public static void output_buffer(int buf[ ], int count) {
    // Output a block of data to the device
    int status, i, ready;
    for (i = 0; i < count; i++) {
        do {
            status = in(display_status_reg);// get status
            ready = (status << 7) & 0x01;// isolate ready bit
        } while (ready == 1);
        out(display_buffer_reg, buf[i]);
    }
}
```

Figure 5-31. An example of programmed I/O.

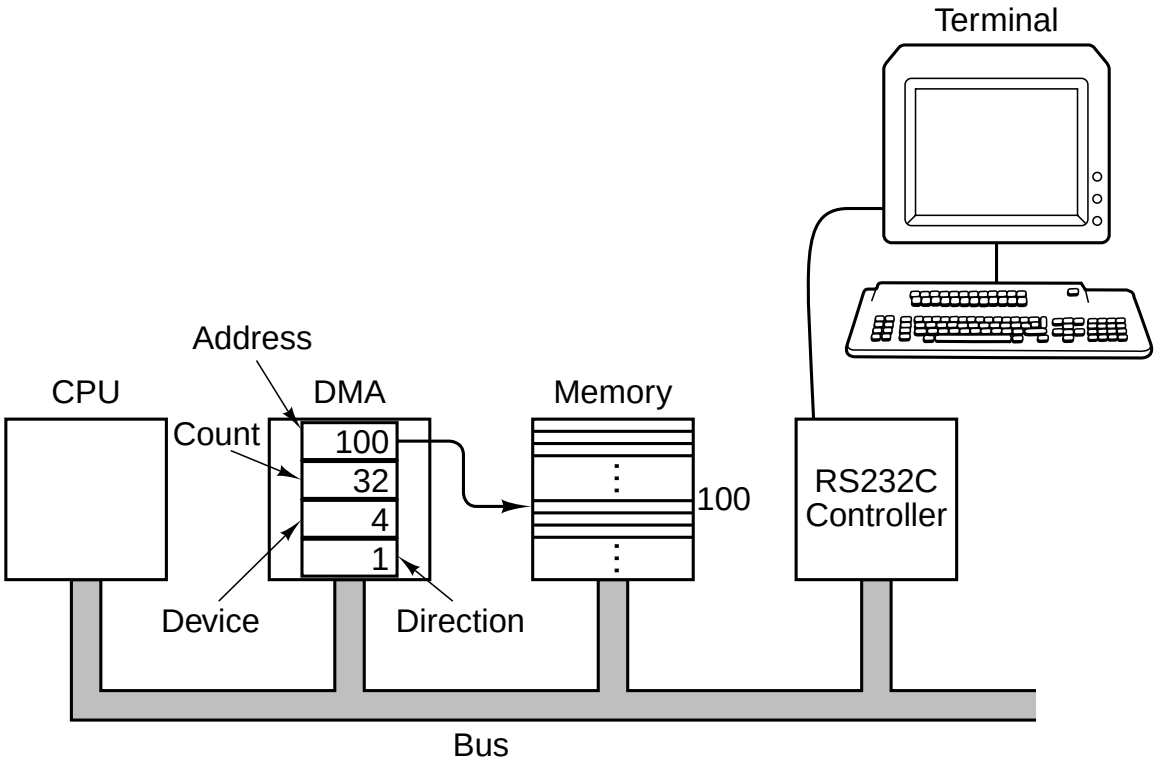


Figure 5-32. A system with a DMA controller.

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOV DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract DST from SRC
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract DST & carry from SRC
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT ADDR	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination
# = shift/rotate count
LV = # locals

Figure 5-33. A selection of the Pentium II integer instructions.

Loads

LDSB ADDR,DST	Load signed byte (8 bits)
LDUB ADDR,DST	Load unsigned byte (8 bits)
LDSH ADDR,DST	Load signed halfword (16 bits)
LDUH ADDR,DST	Load unsigned halfword (16)
LDSW ADDR,DST	Load signed word (32 bits)
LDUW ADDR,DST	Load unsigned word (32 bits)
LDX ADDR,DST	Load extended (64-bits)

Stores

STB SRC,ADDR	Store byte (8 bits)
STH SRC,ADDR	Store halfword (16 bits)
STW SRC,ADDR	Store word (32 bits)
STX SRC,ADDR	Store extended (64 btis)

Arithmetic

ADD R1,S2,DST	Add
ADDCC "	Add and set icc
ADDC "	Add with carry
ADDCCC "	Add with carry and set icc
SUB R1,S2,DST	Subtract
SUBCC "	Subtract and set icc
SUBC "	Subtract with carry
SUBCCC "	Subtract with carry and set icc
MULX R1,S2,DST	Multiply
SDIVX R1,S2,DST	Signed divide
UDIVX R1,S2,DST	Unsigned divide
TADCC R1,S2,DST	Tagged add

Shifts/rotates

SLL R1,S2,DST	Shift left logical (64 bits)
SLLX R1,S2,DST	Shift left logical extended (64)
SRL R1,S2,DST	Shift right logical (32 bits)
SRLX R1,S2,DST	Shift right logical extended (64)
SRA R1,S2,DST	Shift right arithmetic (32 bits)
SRAX R1,S2,DST	Shift right arithmetic ext. (64)

Boolean

AND R1,S2,DST	Boolean AND
ANDCC "	Boolean AND and set icc
ANDN "	Boolean NAND
ANDNCC "	Boolean NAND and set icc
OR R1,S2,DST	Boolean OR
ORCC "	Boolean OR and set icc
ORN "	Boolean NOR
ORNCC "	Boolean NOR and set icc
XOR R1,S2,DST	Boolean XOR
XORCC "	Boolean XOR and set icc
XNOR "	Boolean EXCLUSIVE NOR
XNORCC "	Boolean EXCL. NOR and set icc

Transfer of control

BPcc ADDR	Branch with prediction
BPr SRC,ADDR	Branch on register
CALL ADDR	Call procedure
RETURN ADDR	Return from procedure
JMPL ADDR,DST	Jump and Link
SAVE R1,S2,DST	Advance register windows
RESTORE "	Restore register windows
Tcc CC,TRAP#	Trap on condition
PREFETCH FCN	Prefetch data from memory
LDSTUB ADDR,R	Atomic load/store
MEMBAR MASK	Memory barrier

Miscellaneous

SETHI CON,DST	Set bits 10 to 31
MOVcc CC,S2,DST	Move on condition
MOVr R1,S2,DST	Move on register
NOP	No operation
POPC S1,DST	Population count
RDCCR V,DST	Read condition code register
WRCCR R1,S2,V	Write condition code register
RDPC V,DST	Read program counter

SRC = source register
DST = destination register
R1 = source register
S2 = source: register or immediate
ADDR = memory address

TRAP# = trap number
FCN = function code
MASK = operation type
CON = constant
V = register designator

CC = condition code set
R =destination register

cc = condition
r = LZ,LEZ,Z,NZ,GZ,GEZ

Figure 5-34. The primary UltraSPARC II integer instructions.

Instruction	How to do it
MOV SRC,DST	OR SRC with G0 and store the result DST
CMP SRC1,SRC2	SUBCC SRC2 from SRC1 and store the result in G0
TST SRC	ORCC SRC1 with G0 and store the result in G0
NOT DST	XNOR DST with G0
NEG DST	SUB DST from G0 and store in DST
INC DST	ADD 1 to DST (immediate operand)
DEC DST	SUB 1 from DST (immediate operand)
CLR DST	OR G0 with G0 and store in DST
NOP	SETHI G0 to 0
RET	JMPL %I7+8,%G0

Figure 5-35. Some simulated UltraSPARC II instructions.

Loads

typeLOAD IND8	Push local variable onto stack
typeALOAD	Push array element on stack
BALOAD	Push byte from an array on stack
SALOAD	Push short from an array on stack
CALOAD	Push char from an array on stack
AALOAD	Push pointer from an array on ”

Stores

typeSTORE IND8	Pop value and store in local var
typeASTORE	Pop value and store in array
BASTORE	Pop byte and store in array
SASTORE	Pop short and store in array
CASTORE	Pop char and store in array
AASTORE	Pop pointer and store in array

Pushes

BIPUSH CON8	Push a small constant on stack
SIPUSH CON16	Push 16-bit constant on stack
LDC IND8	Push constant from const pool
typeCONST_#	Push immediate constant
ACONST_NULL	Push a null pointer on stack

Arithmetic

typeADD	Add
typeSUB	Subtract
typeMUL	Multiple
typeDIV	Divide
typeREM	Remainder
typeNEG	Negate

Boolean/shift

iIAND	Boolean AND
iIOR	Boolean OR
iIXOR	Boolean EXCLUSIVE OR
iISHL	Shift left
iISHR	Shift right
iIUSHR	Unsigned shift right

Conversion

x2y	Convert x to y
i2c	Convert integer to char
i2b	Convert integer to byte

Stack management

DUPxx	Six instructions for duping
POP	Pop an int from stk and discard
POP2	Pop two ints from stk and discard
SWAP	Swap top two ints on stack

Comparison

IF_ICMPrel OFFSET16	Conditional branch
IF_ACMPEQ OFFSET16	Branch if two ptrs equal
IF_ACMPNE OFFSET16	Branch if ptrs unequal
IFrel OFFSET16	Test 1 value and branch
IFNULL OFFSET16	Branch if ptr is null
IFNONNULL OFFSET16	Branch if ptr is nonnull
LCMP	Compare two longs
FCMPL	Compare 2 floats for <
FCMPG	Compare 2 floats for >
DCMPL	Compare doubles for <
DCMPG	Compare doubles for >

Transfer of control

INVOKEVIRTUAL IND16	Method invocation
INVOKESTATIC IND16	Method invocation
INVOKEINTERFACE ...	Method invocation
INVOKESPECIAL IND16	Method invocation
JSR OFFSET16	Invoke finally clause
typeRETURN	Return value
ARETURN	Return pointer
RETURN	Return void
RET IND8	Return from finally
GOTO OFFSET16	Unconditional branch

Arrays

ANEWARRAY IND16	Create array of ptrs
NEWARRAY ATYPE	Create array of atype
MULTINEWARRAY IN16,D	Create multidim array
ARRAYLENGTH	Get array length

Miscellaneous

IINC IND8,CON8	Increment local variable
WIDE	Wide prefix
NOP	No operation
GETFIELD IND16	Read field from object
PUTFIELD IND16	Write field to object
GETSTATIC IND16	Get static field from class
NEW IND16	Create a new object
INSTANCEOF OFFSET16	Determine type of obj
CHECKCAST IND16	Check object type
ATHROW	Throw exception
LOOKUPSWITCH ...	Sparse multiway branch
TABLESWITCH ...	Dense multiway branch
MONITORENTER	Enter a monitor
MONITOREXIT	Leave a monitor

IND8/16 = index of local variable type, x, y = I, L, F, D
CON8/16, D, ATYPE = constant OFFSET16 for branch

Figure 5-36. The JVM instruction set.

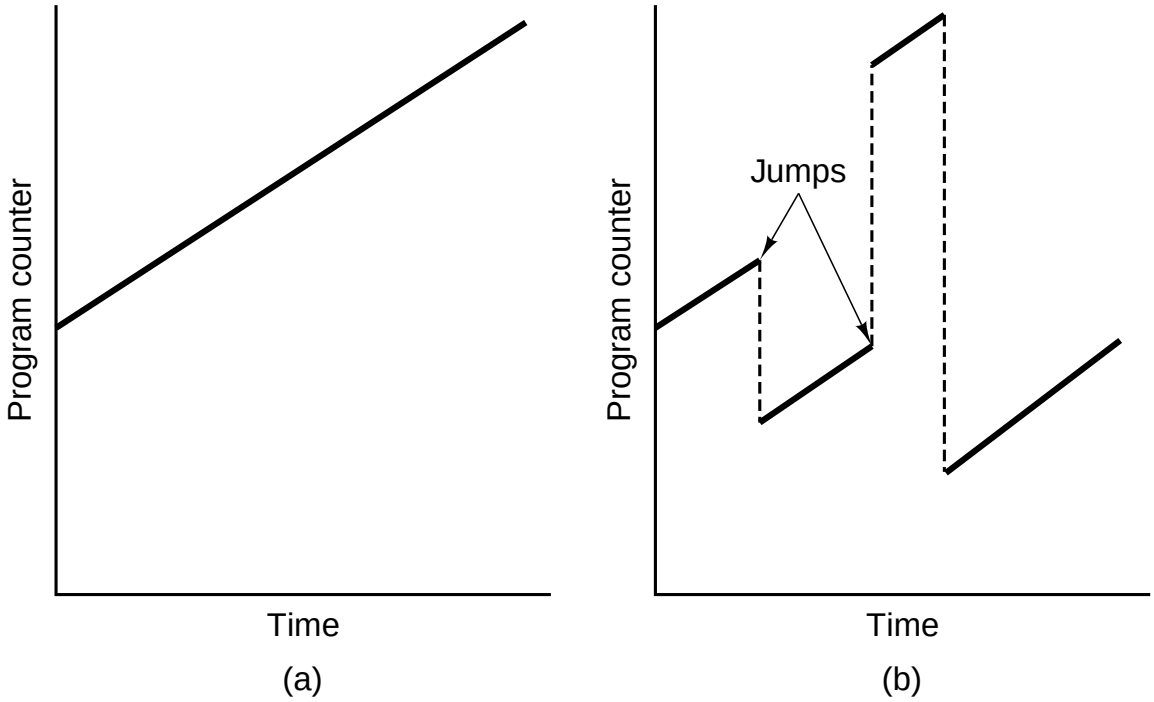


Figure 5-37. Program counter as a function of time (smoothed). (a) Without branches. (b) With branches.

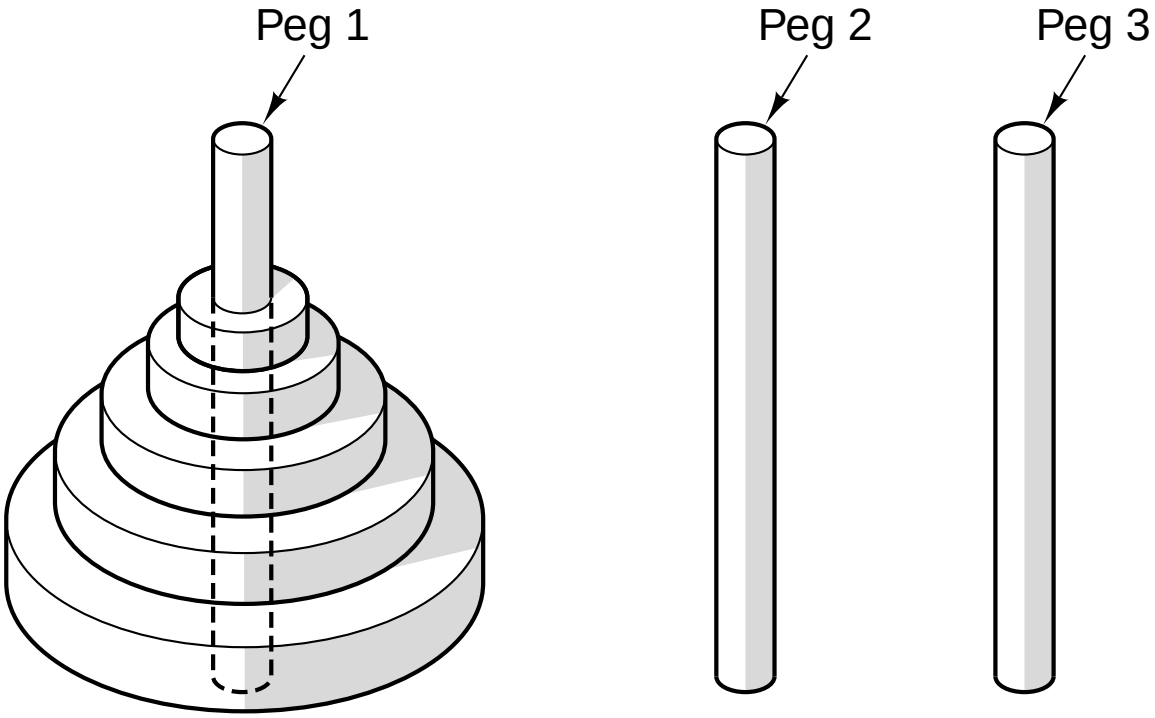


Figure 5-38. Initial configuration for the Towers of Hanoi problem for five disks.

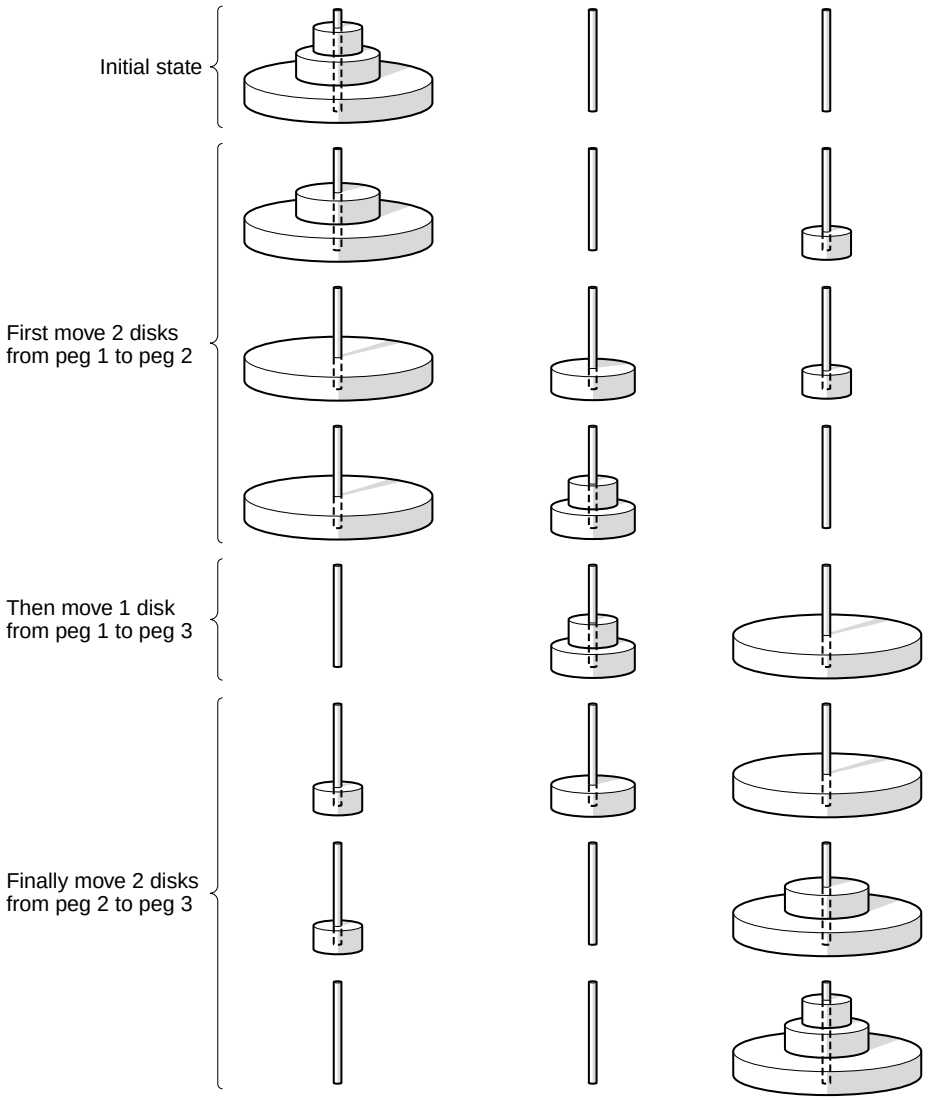


Figure 5-39. The steps required to solve the Towers of Hanoi for three disks.

```
public void towers(int n, int i, int j) {
    int k;

    if (n == 1)
        System.out.println("Move a disk from " + i + " to " + j);
    else {
        k = 6 – i – j;
        towers(n – 1, i, k);
        towers(1, i, j);
        towers(n – 1, k, j);
  }
}
```

Figure 5-40. A procedure for solving the Towers of Hanoi.

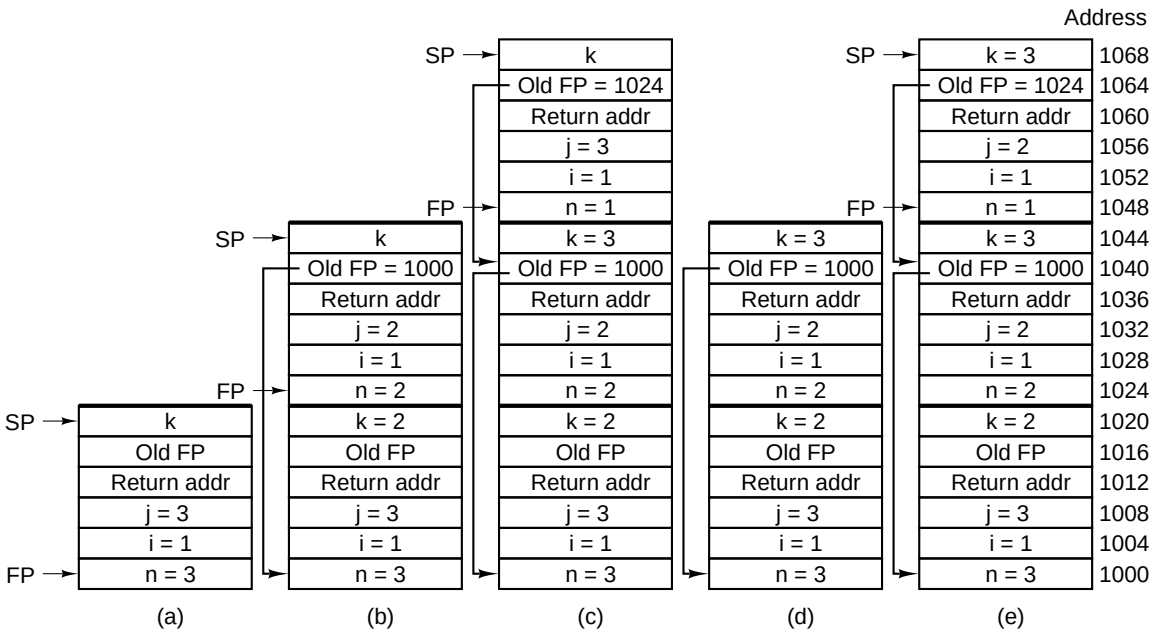


Figure 5-41. The stack at several points during the execution of Fig. 5-40.

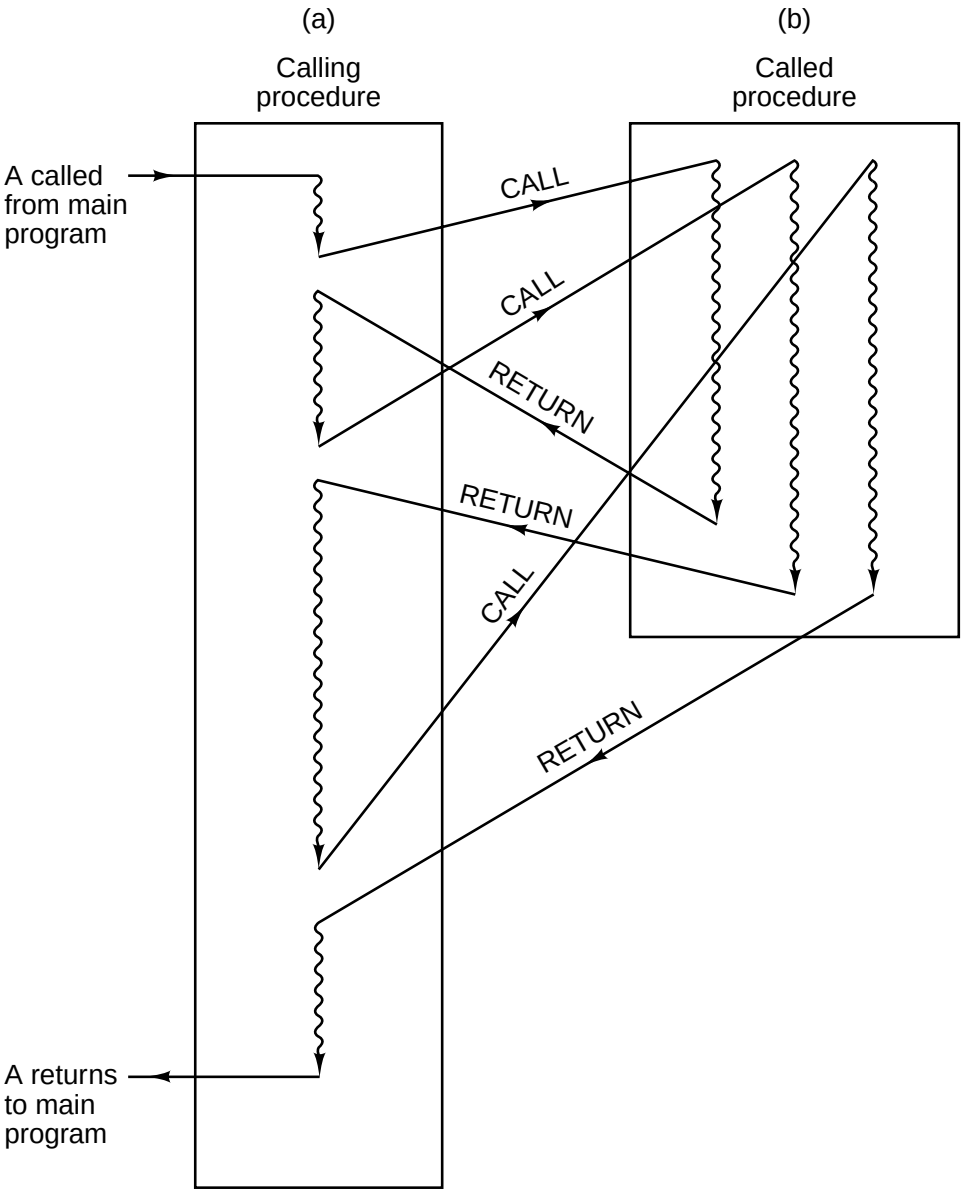


Figure 5-42. When a procedure is called, execution of the procedure always begins at the first statement of the procedure.

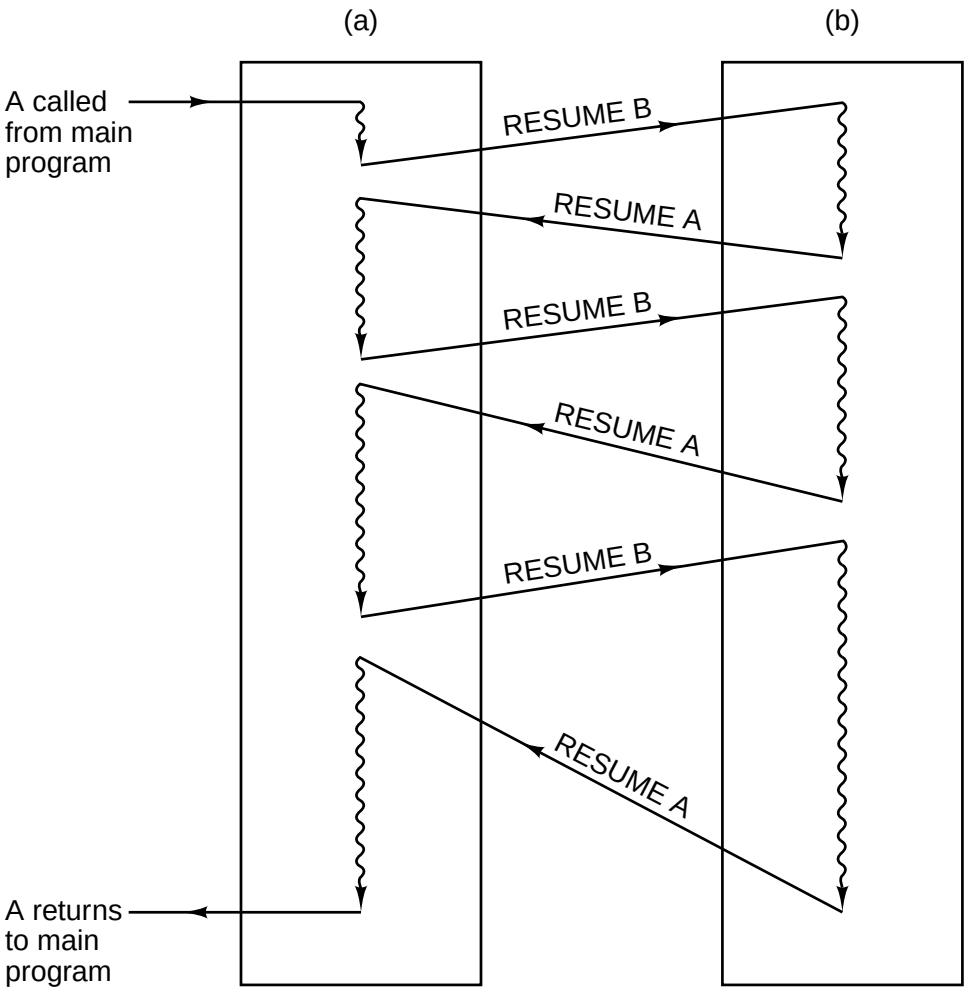


Figure 5-43. When a coroutine is resumed, execution begins at the statement where it left off the previous time, not at the beginning.

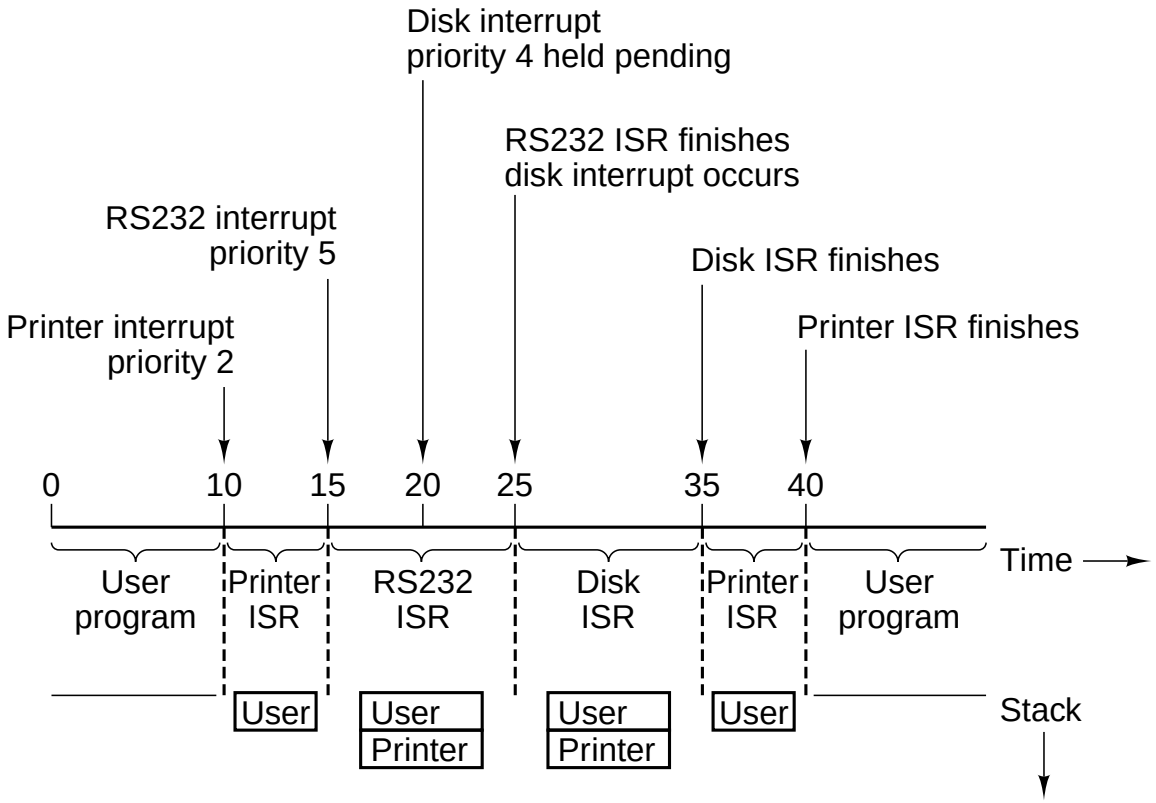


Figure 5-44. Time sequence of multiple interrupt example.

```
        .586                    ; compile for Pentium (as opposed to 8088 etc.)
.MODEL FLAT
PUBLIC _towers                  ; export 'towers'
EXTERN _printf:NEAR             ; import printf
.CODE
 _towers:                       PUSH EBP; save EBP (frame pointer)
        MOV EBP, ESP            ; set new frame pointer above ESP
        CMP [EBP+8], 1          ; if (n == 1)
        JNE L1                  ; branch if n is not 1
        MOV EAX, [EBP+16]       ; printf(" ...", i, j);
        PUSH EAX                ; note that parameters i, j and the format
        MOV EAX, [EBP+12]       ; string are pushed onto the stack
        PUSH EAX                ; in reverse order. This is the C calling convention
        PUSH OFFSET FLAT:format; offset flat means the address of format
        CALL _printf            ; call printf
        ADD ESP, 12             ; remove params from the stack
        JMP Done                ; we are finished
 L1:    MOV EAX, 6              ; start k = 6 – i – j
        SUB EAX, [EBP+12]       ; EAX = 6 – i
        SUB EAX, [EBP+16]       ; EAX = 6 – i – j
        MOV [EBP+20], EAX       ; k = EAX
        PUSH EAX                ; start towers(n – 1, i, k)
        MOV EAX, [EBP+12]       ; EAX = i
        PUSH EAX                ; push i
        MOV EAX, [EBP+8]        ; EAX = n
        DEC EAX                 ; EAX = n – 1
        PUSH EAX                ; push n – 1
        CALL _towers            ; call towers(n – 1, i, 6 – i – j)
        ADD ESP, 12             ; remove params from the stack
        MOV EAX, [EBP+16]       ; start towers(1, i, j)
        PUSH EAX                ; push j
        MOV EAX, [EBP+12]       ; EAX = i
        PUSH EAX                ; push i
        PUSH 1                  ; push 1
        CALL _towers            ; call towers(1, i, j)
        ADD ESP, 12             ; remove params from the stack
        MOV EAX, [EBP+12]       ; start towers(n – 1, 6 – i – j, i)
        PUSH EAX                ; push i
        MOV EAX, [EBP+20]       ; push 20
        PUSH EAX                ; push k
        MOV EAX, [EBP+8]        ; EAX = n
        DEC EAX                 ; EAX = n–1
        PUSH EAX                ; push n – 1
        CALL _towers            ; call towers(n – 1, 6 – i – j, i)
        ADD ESP, 12             ; adjust stack pointer
 Done:  LEAVE                   ; prepare to exit
        RET 0                   ; return to the caller
.DATA
format  DB "Move disk from %d to %d\n"; format string
END
```

Figure 5-45. The Towers of Hanoi for the Pentium II.

```
#define N %i0                   /* N is input parameter 0 */
#define I %i1                   /* I is input parameter 1 */
#define J %i2                   /* J is input parameter 2 */
#define K %l0                   /* K is local variable 0 */
#define Param0 %o0              /* Param0 is output parameter 0 */
#define Param1 %o1              /* Param1 is output parameter 1 */
#define Param2 %o2              /* Param2 is output parameter 2 */
#define Scratch %l1             /* as an aside, cpp uses the C comment convention */
        .proc 04
        .global towers

towers:                         save %sp, –112, %sp
        cmp N, 1                ! if (n == 1)
        bne Else                ! if (n != 1) goto Else

        sethi %hi(format), Param0 ! printf("Move a disk from %d to %d\n", i, j)
        or Param0, %lo(format), Param0! Param0 = address of format string
        mov I, Param1           ! Param1 = i
        call printf             ! call printf BEFORE parameter 2 (j) is set up
        mov J, Param2           ! use the delay slot after call to set up parameter 2
        b Done                  ! we are done now
        nop                     ! fill delay slot


Else:   mov 6, K                ! start k = 6 –i – j
        sub K, J, K             ! k = 6 – j
        sub K, I, K             ! k = 6 – i – j

        add N, –1, Scratch      ! start towers(n – 1, i, k)
        mov Scratch, Param0     ! Scratch = N – 1
        mov I, Param1           ! parameter 1 = i
        call towers             ! call towers BEFORE parameter 2 (k) is set up
        mov K, Param2           ! use the delay slot after call to set up parameter 2

        mov 1, Param0           ! start towers(1, i, j)
        mov I, Param1           ! parameter 1 = i
        call towers             ! call towers BEFORE parameter 2 (j) is set up
        mov J, Param2           ! parameter 2 = j

        mov Scratch, Param0     ! start towers(n – 1, k, j)
        mov K, Param1           ! parameter 1 = k
        call towers             ! call towers BEFORE parameter 2 (j) is set up
        mov J, Param2           ! parameter 2 = j

Done:   ret                     ! return
        restore                 ! use the delay slot after ret to restore windows

format:                         .asciz "Move a disk from %d to %d\n"
```

Figure 5-46. The Towers of Hanoi for the UltraSPARC II.

```
        ILOAD_0             // local 0 = n; push n
        ICONST_1            // push 1
        IF_ICMPNE L1        // if (n != 1) goto L1

        GETSTATIC #13       // n == 1; this code handles the println statement
        NEW #7              // allocate buffer for the string to be built
        DUP                 // duplicate the pointer to the buffer
        LDC #2              // push pointer to string "move a disk from "
        INVOKESPECIAL #10   // copy the string to the buffer
        ILOAD_1             // push i
        INVOKEVIRTUAL #11   // convert i to string and append to the new buffer
        LDC #1              // push pointer to string " to "
        INVOKEVIRTUAL #12   // append this string to the buffer
        ILOAD_2             // push j
        INVOKEVIRTUAL #11   // convert j to string and append to buffer
        INVOKEVIRTUAL #15   // string conversion
        INVOKEVIRTUAL #14   // call println
        RETURN              // return from towers

L1:     BIPUSH 6            // Else part: compute k = 6 – i – j
        ILOAD_1             // local 1 = i; push i
        ISUB                // top-of-stack = 6 – i
        ILOAD_2             // local 2 = j; push j
        ISUB                // top-of-stack = 6 – i – j
        ISTORE_3            // local 3 = k = 6 – i – j; stack is now empty

        ILOAD_0             // start working on towers(n – 1, i, k); push n
        ICONST_1            // push 1
        ISUB                // top-of-stack = n – 1
        ILOAD_1             // push i
        ILOAD_3             // push k
        INVOKESTATIC #16    // call towers(n – 1, 1, k)

        ICONST_1            // start working on towers(1, i, j); push 1
        ILOAD_1             // push i
        ILOAD_2             // push j
        INVOKESTATIC #16    // call towers(1, i, j)

        ILOAD_0             // start working on towers(n – 1, k, j); push n
        ICONST_1            // push 1
        ISUB                // top-of-stack = n – 1
        ILOAD_3             // push k
        ILOAD_2             // push j
        INVOKESTATIC #16    // call towers(n – 1, k, j)
        RETURN              // return from towers
```

Figure 5-47. The Towers of Hanoi for JVM.

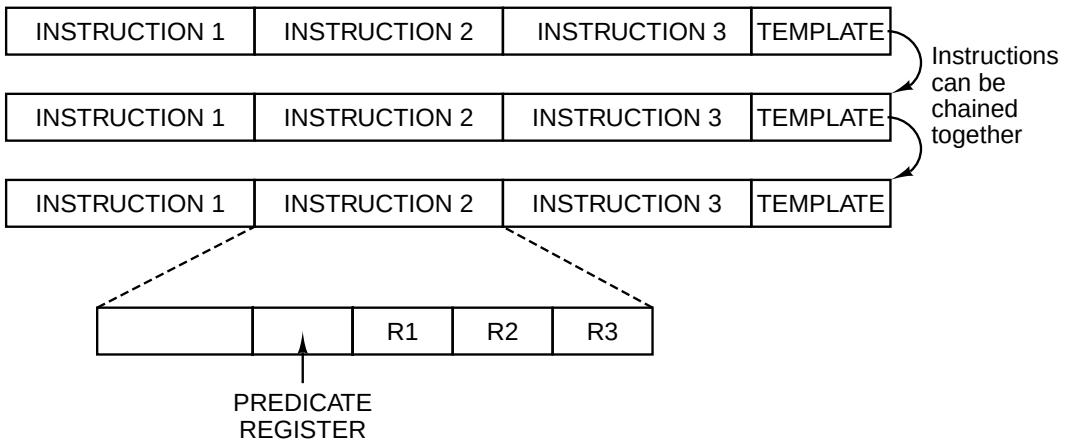


Figure 5-48. IA-64 is based on bundles of three instructions.

```
if (R1 == 0)              CMP R1,0          CMOVZ R2,R3,R1
    R2 = R3;              BNE L1
                          MOV R2,R3
                    L1:
```

(a) (b) (c)

Figure 5-49. (a) An if statement. (b) Generic assembly code for (a). (c) A conditional instruction.

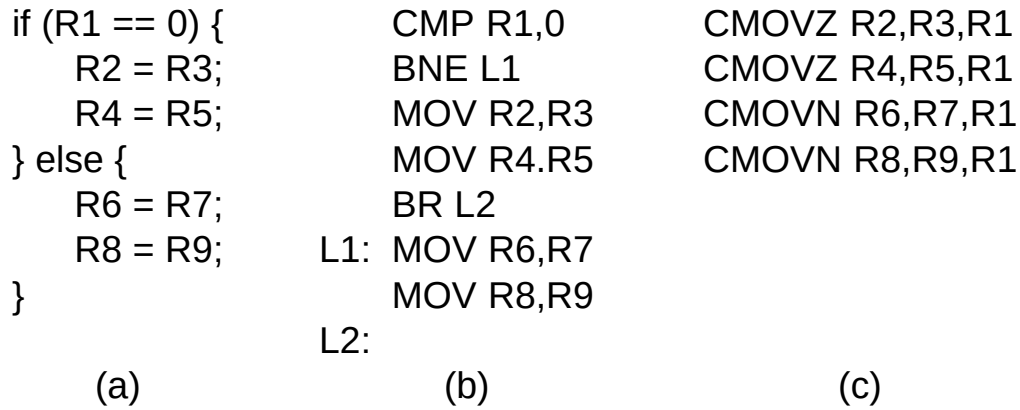


Figure 5-50. (a) An if statement. (b) Generic assembly code for (a). (c) Conditional execution.

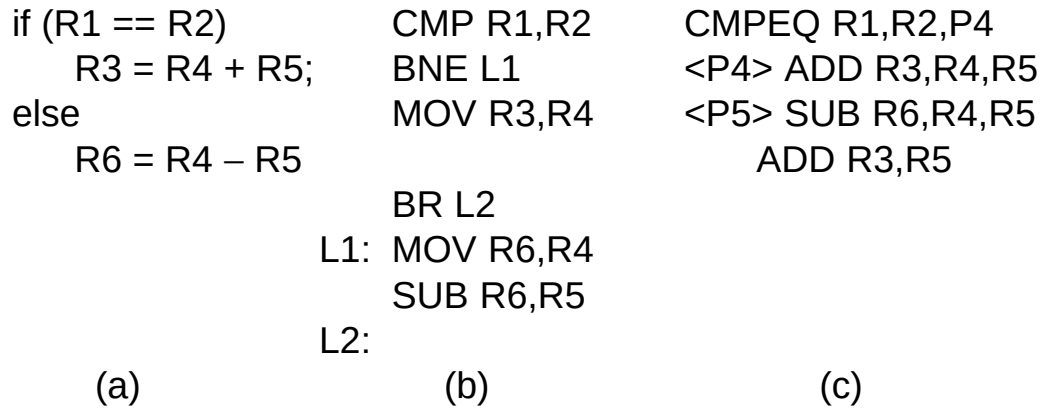


Figure 5-51. (a) An if statement. (b) Generic assembly code for (a). (c) Predicated execution.

5.8 THE INTEL IA-64

Intel is rapidly getting to the point where it has just about squeezed every last drop of juice out of the IA-32 ISA and the Pentium II line of processors. New models can still benefit from advances in manufacturing technology, which means smaller transistors (hence faster clock speeds). However, finding new tricks to speed up the implementation even more is getting harder and harder as the constraints imposed by the IA-32 ISA are looming larger all the time.

The only real solution is to abandon the IA-32 as the main line of development and go to a completely new ISA. This is, in fact, what Intel intends to do. The new architecture, developed jointly by Intel and Hewlett Packard, is called the **IA-64**. It is a full 64-bit machine from beginning to end. A whole series of CPUs implementing this architecture is expected in the coming years. The first implementation, code named the **Merced** (after somebody's favorite California river), is a high-end CPU, but eventually there will be a whole spectrum of CPUs, from high-end servers to entry-level desktop models.

Given the importance to the computer industry of anything Intel does, it is worth taking a look at the IA-64 architecture (and by implication, the Merced). However, that aside, the key ideas beyond the IA-64 are already well known among researchers in the field and may well surface in other designs as well. In fact, some of them are already present in various forms in (experimental) systems. In the following sections we will look at the nature of the problem Intel is facing, the model that IA-64 adopts to deal with it, how some of the key ideas work, and what might happen.

5.8.1 The Problem with the Pentium II

The main fact of life that causes all the trouble is that IA-32 is an ancient ISA with all the wrong properties for current technology. It is a CISC ISA with variable-length instructions and a myriad of different formats that are hard to decode quickly on the fly. Current technology works best with RISC ISAs that have one instruction length and a fixed length opcode that is easy to decode. The IA-32 instructions can be broken up into RISC-like micro-operations at execution time, but doing so requires hardware (chip area), takes time, and adds complexity to the design. That is strike one.

The IA-32 is also a two-address memory-oriented ISA. Most instructions reference memory, and most programmers and compilers think nothing of referencing memory all the time. Current technology favors load/store ISAs that only reference memory to get the operands into registers but otherwise perform all their calculations using three-address memory register instructions. And with CPU clock speeds going up much faster than memory speeds, the problem will get worse with time. That is strike two.

The IA-32 also has a small and irregular register set. Not only does this tie compilers in knots, but the small number of general-purpose registers (four or six, depending on how you count ESI and EDI) requires intermediate results to be spilled into memory all the time, generating extra memory references even when they are not logically needed. That is strike three. The IA-32 is out.

Now let us start the second inning. The small number of registers causes many dependences, especially unnecessary WAR dependences, because results have to go somewhere and there are no extra registers available. Getting around the lack of registers requires the implementation to do renaming internally—a terrible hack if ever there was one—to secret registers inside the reorder buffer. To avoid blocking on cache misses too often, instructions have to be executed out of order. However, the IA-32's semantics specify precise interrupts, so the out-of-order instructions have to be retired in order. All of these things require a lot of very complex hardware. Strike four.

Doing all this work quickly requires a very deep (12-stage) pipeline. In turn, the deep pipeline means that instructions are entered into it 11 cycles before they will be finished. Consequently, very accurate branch prediction is essential to make sure the right instructions are being entered into the pipeline. Because a misprediction requires the pipeline to be flushed, at great cost, even a fairly low misprediction rate can cause a substantial performance degradation. Strike five.

To alleviate the problems with mispredictions, the processor has to do speculative execution, with all the problems that entails, especially when memory references on the wrong path cause an exception. Strike six.

We are not going to play the whole baseball game here, but it should be clear by now that there is a real problem. And we have not even mentioned the fact that IA-32's 32-bit addresses limit individual programs to 4 GB of memory, which is a growing problem on high-end servers.

All in all, the situation with IA-32 can be favorably compared to the state of celestial mechanics just prior to Copernicus. The then-current theory dominating astronomy was that the earth was fixed and motionless in space and that the planets moved in circles with epicycles around it. However, as observations got better and more deviations from this model could be clearly observed, epicycles were added to the epicycles until the whole model just collapsed from its internal complexity

Intel is in the same pickle now. A huge fraction of all the transistors on the Pentium II are devoted to decomposing CISC instructions, figuring out what can be done in parallel, resolving conflicts, making predictions, repairing the consequences of incorrect predictions and other bookkeeping, leaving surprisingly few over for doing the real work the user asked for. The conclusion that Intel is being inexorably driven to is the only sane conclusion: junk the whole thing (IA-32) and start all over with a clean slate (IA-64).

5.8.2 The IA-64 Model: Explicitly Parallel Instruction Computing

The starting point for IA-64 was a high-end 64-bit RISC processor (of which the UltraSPARC II is one of many current examples). Since the IA-64 was designed jointly with Hewlett Packard, no doubt HP's PA-RISC architecture was the real basis, but all state-of-the-art RISC machines are similar enough that our discussion of the UltraSPARC II is still highly germane. The Merced is a dual-mode processor, with the capability of running both IA-32 and IA-64 programs, but we will focus only on the IA-64 part in the discussion below.

The IA-64 architecture is a load/store architecture with 64-bit addresses and 64-bit wide registers. There are 64 general registers available to IA-64 programs (and additional registers available to IA-32 programs). All instructions have the same fixed format: an opcode, two 6-bit source register fields, a 6-bit destination register field, and another 6-bit field to be discussed later on. Most instructions take two register operands, perform some computation on them, and put the result back in the destination register. Many functional units are available for doing different operations in parallel. So far, nothing unusual. Most RISC machines have an architecture pretty much like this.

What is unusual is the idea of a bundle of related instructions. Instructions come in groups of three, called a **bundle**, as shown in Fig. 5-1. Each 128-bit bit bundle contains three 40-bit fixed-format instructions and an 8-bit template. Bundles can be chained together using an end-of-bundle bit, so more than three instructions can be present in one bundle. The template contains information about which instructions can be executed in parallel. This scheme, plus the existence of so many registers, allows the compiler to isolate blocks of instructions and tell the CPU that they can be executed in parallel. Thus the compiler must reorder instructions, check for dependences, make sure there are functional units available, etc., instead of the hardware. The idea is that by exposing the internal workings of the machine and telling compiler writers to make sure that each bundle consists of compatible instructions, the job of scheduling the RISC instructions is moved from the hardware (at run time) to the compiler (at compile time). For this reason, the model is called **EPIC** (**Explicitly Parallel Instruction Computing**).

There are several advantages to doing instruction scheduling at compile time. First, since the compiler is now doing all the work, the hardware can be much simpler, saving millions of transistors for other useful functions, such as larger level 1 caches. Second, for any given program, the scheduling has to be done only once, at compile time. Third, since the compiler is now doing all the work, it will be possible for a software vendor to use a compiler that spends hours optimizing its program and have all the users have the benefit every time the program is run.

The idea of bundles of instructions can be used to create a whole family of processors. On the low-end CPUs, one bundle may be issued per clock cycle.

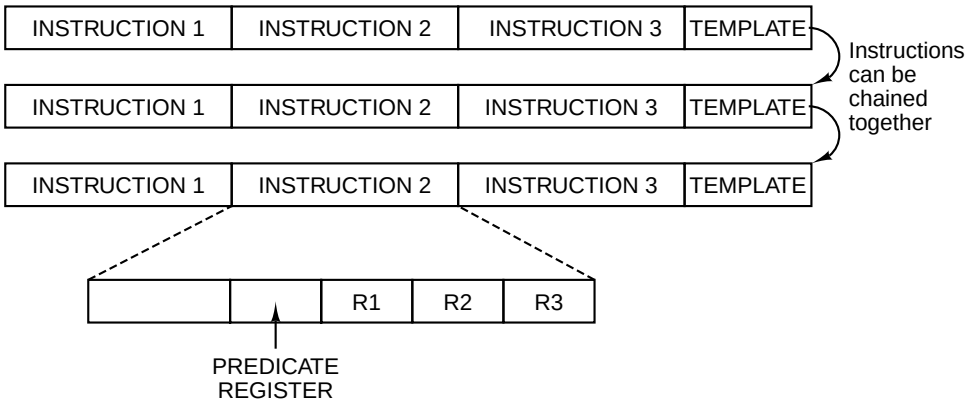


Figure 5-1. IA-64 is based on bundles of three instructions.

The CPU may wait until all the instructions have completed before issuing the next bundle. On high-end CPUs, it may be possible to issue multiple bundles during the same clock cycle, analogous to current superscalar designs. Also, on high-end CPUs, the processor may begin issuing instructions from a new bundle before all instructions from the previous bundle have completed. Of course, it needs to check to see if the registers and functional unit needed are available, but it does not have to check to see if other instructions from its own bundle are in conflict with it as the compiler has guaranteed that this is not the case.

5.8.3 Predication

Another important feature of IA-64 is the new way it deals with conditional branches. If there were a way to get rid of most of them, CPUs could be made much simpler and faster. At first thought it might seem that getting rid of conditional branches would be impossible because programs are full of if statements. However, IA-64 uses a technique called **predication** that can greatly reduce their number (August et al., 1998; and Hwu, 1998). We will now briefly describe it.

In current machines, all instructions are unconditional in the sense that when the CPU hits an instruction, it just carries the instruction out. There is no internal debate of the form: ‘‘To do or not to do, that is the question?’’ In contrast, in a predicated architecture, instructions contain conditions (predicates) telling when they should be executed and when not. This paradigm shift from unconditional instructions to predicated instructions is what makes it possible to get rid of (many) conditional branches. Instead of having to make a choice between one sequence of unconditional instructions or another sequence of unconditional instructions, all the instructions are merged into a single sequence of predicated

instructions, using different predicates for different instructions.

To see how predication works, let us start with the simple example of Fig. 5-2, which shows **conditional execution**, a precursor to predication. In Fig. 5-2(a) we see an if statement. In Fig. 5-2(b) we see its translation into three instructions: a comparison, a conditional branch, and a move instruction.

```
if (R1 == 0)          CMP R1,0          CMOVZ R2,R3,R1
    R2 = R3;          BNE L1
                      MOV R2,R3
                  L1:
```

(a) (b) (c)

Figure 5-2. (a) An if statement. (b) Generic assembly code for (a). (c) A conditional instruction.

In Fig. 5-2(c) we get rid of the conditional branch by using a new instruction, CMOVZ, which is a conditional move. What it does is check to see if the third register, R1, is 0. If so, it copies R3 to R2. If not, it does nothing.

Once we have an instruction that can copy data when some register is 0, it is a small step to an instruction that can copy data when some register is not 0, say CMOVN. With both of these instructions available, we are on our way to full conditional execution. Imagine an if statement with several assignments in the then part and several assignments in the else part. The whole statement can be translated into code to set some register to 0 if the condition is false and another value if it is true. Following the register setup, the then part assignments can be compiled into a sequence of CMOVN instructions and the else part assignments could be compiled into a sequence of CMOVZ instructions.

All of these instructions, the register setup, the CMOVNs and the CMOVZs form a single basic block with no conditional branch. The instructions can even be reordered, either by the compiler (including hoisting the assignments before the test), or during execution. The only catch is that the condition has to be known by the time the conditional instructions have to be retired (near the end of the pipeline). A simple example showing a then part and an else part is given in Fig. 5-3.

Although we have shown only very simple conditional instructions here (taken from the Pentium II, actually), on the IA-64 all instructions are predicated. What this means is that the execution of every instruction can be made conditional. The extra 6-bit field referred to earlier selects one of 64 1-bit predicate registers. Thus an if statement will be compiled into code that sets one of the predicate registers to 1 if the condition is true and to 0 if it is false. Simultaneously and automatically, it sets another predicate register to the inverse value. Using predication, the machine instructions forming the then and else clauses will be merged into a single stream of instructions, the former ones using the predicate and the latter ones using its inverse.

```
if (R1 == 0) {
    R2 = R3;
    R4 = R5;
} else {
    R6 = R7;
    R8 = R9;
}
```

(a)

```
    CMP R1,0
    BNE L1
    MOV R2,R3
    MOV R4.R5
    BR L2
L1: MOV R6,R7
    MOV R8,R9
L2:
```

(b)

```
CMOVZ R2,R3,R1
CMOVZ R4,R5,R1
CMOVN R6,R7,R1
CMOVN R8,R9,R1
```

(c)

Figure 5-3. (a) An if statement. (b) Generic assembly code for (a). (c) Conditional execution.

Although simple, the example of Fig. 5-4 shows the basic idea of how predication can be used to eliminate branches. The CMPEQ instruction compares two registers and sets the predicate register P4 to 1 if they are equal and to 0 if they are different. If also sets a paired register, say, P5, to the inverse condition. Now the instructions for the if and then parts can be put after one another, each one conditioned on some predicate register (shown in angle brackets). Arbitrary code can be put here provided that each instruction is properly predicated.

```
if (R1 == R2)
    R3 = R4 + R5;
else
    R6 = R4 – R5
```

(a)

```
    CMP R1,R2
    BNE L1
    MOV R3,R4
    ADD R3,R5
    BR L2
L1: MOV R6,R4
    SUB R6,R5
L2:
```

(b)

```
CMPEQ R1,R2,P4
<P4> ADD R3,R4,R5
<P5> SUB R6,R4,R5
```

(c)

Figure 5-4. (a) An if statement. (b) Generic assembly code for (a). (c) Predicated execution.

In the IA-64, this idea is taken to the extreme, with comparison instructions for setting the predicate registers as well as arithmetic and other instructions whose execution is dependent on some predicate register. Predicated instructions can be stuffed into the pipeline in sequence, with no stalls and no problems. That is why they are so useful.

The way predication really works on the IA-64 is that every instruction is actually executed. At the very end of the pipeline, when it is time to retire an instruction, a check is made to see if the predicate is true. If so, the instruction is retired normally and its results are written back to the destination register. If the predicate is false, no writeback is done so the instruction has no effect.

Predication is discussed further in (Dulong, 1998).

5.8.4 Speculative Loads

Another feature of the IA-64 that speeds up execution is the presence of speculative LOADs. If a LOAD is speculative and it fails, instead of causing an exception, it just stops and a bit associated with the register to be loaded is set marking the register as invalid. This is just the poison bit introduced in Chap. 4. If it turns out that the poisoned register is later used, the exception occurs at that time; otherwise, it never happens.

The way speculation is normally used is for the compiler to hoist LOADs to positions before they are needed. By starting early, they may be finished before the results are needed. At the place where the compiler needs to use the register just loaded, it inserts a CHECK instruction. If the value is there, CHECK acts like a NOP and execution continues immediately. If the value is not there yet, the next instruction must stall. If an exception occurred and the poison bit is on, the pending exception occurs at that point.

In summary, a machine implementing the the IA-64 architecture gets its speed from several sources. At the core is a state-of-the-art pipelined, load/store, three-address RISC engine. In addition, IA-64 has a model of explicit parallelism that requires the compiler to figure out which instructions can be executed at the same time without conflicts and group them together in bundles. In this way the CPU can just blindly schedule a bundle without having to do any heavy thinking. Next, predication allows the statements in both branches of an if statement to be merged together in a single stream, eliminating the conditional branch and thus the prediction of which way it will go. Finally, speculative LOADs make it possible to fetch operands in advance, without penalty if it turns out later that they are not needed after all.

5.8.5 Reality Check

If all of these things work as expected, the Merced will indeed be a powerful machine. However, some caveats are in order. First, such an advanced machine has never been built before, certainly not for a mass market audience. History shows that the best laid plans of rodents and persons sometimes go awry for various reasons.

Second, compilers for IA-64 will have to be written. Writing a good compiler for the IA-64 architecture will not be easy. In addition, for the past 30 years, the holy grail of research in parallel programming has been the automatic extraction of parallelism from sequential programs. This work has not been very successful. If a program has little inherent parallelism or the compiler cannot extract it, the IA-64 bundles will all be short, with little gain.

Third, all of the above assumes the operating system is fully 64 bit. Windows 95 and Windows 98 certainly are not and probably never will be. This means that everyone will have to switch to Windows NT or some flavor of UNIX. This transition may not be painless. After all, 10 years after the 32-bit 386 was introduced, Windows 95 still contained a large amount of 16-bit code, and it never used the segmentation hardware that has been in all Intel CPUs for over a decade (segmentation will be discussed in Chap. 6). How long it will take for the operating systems to become fully 64 bit is not known.

Fourth, many people will judge the IA-64 by how well it runs old 16-bit MS-DOS games. When the Pentium Pro came out, it got a lot of bad press because it ran old 16-bit programs no faster than the regular Pentium. Since old 16-bit (or even 32-bit) programs will not even use the new features of IA-64 CPUs, all the predicate registers in the world will not help them. To see any improvement, people will have to buy upgrades of all their software compiled with new IA-64-mode compilers. Many of them will balk at the expense.

Last, there may be alternatives from other vendors (including Intel) that also provide high performance using more conventional RISC architectures, possibly with more conditional instructions, essentially a weaker form of predication. Faster versions of the Pentium II line itself will also be a serious competitor to the IA-64. It is likely that many years will pass before the IA-64 has the kind of market dominance (if ever) that the IA-32 has now.

6

THE OPERATING SYSTEM

MACHINE LEVEL

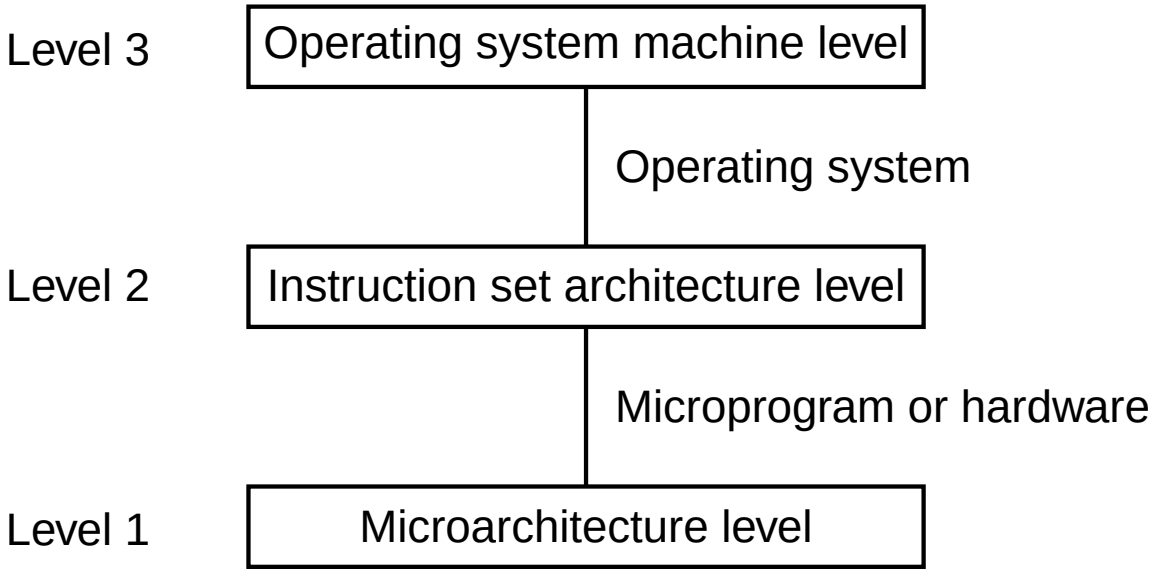


Figure 6-1. Positioning of the operating system machine level.

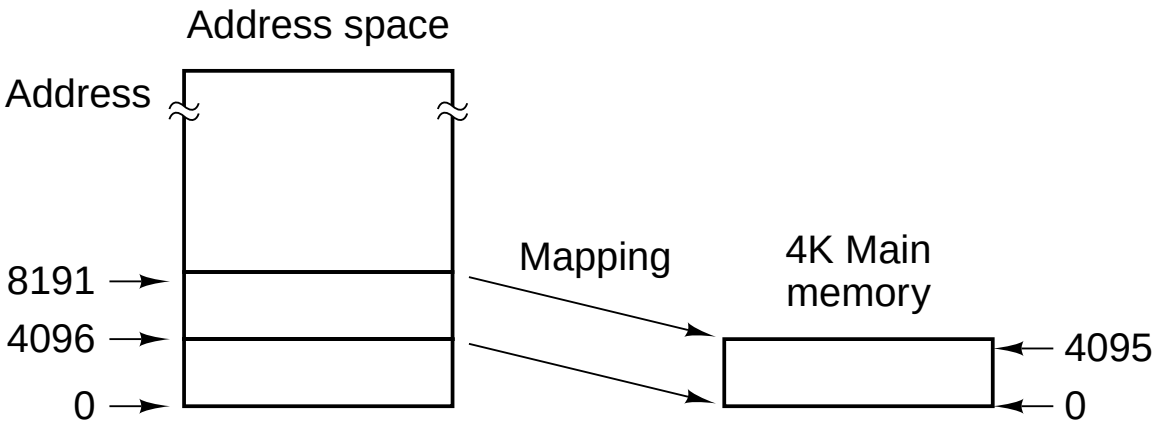


Figure 6-2. A mapping in which virtual addresses 4096 to 8191 are mapped onto main memory addresses 0 to 4095.

Page	Virtual addresses
≈	≈ ≈
15	61440 – 65535
14	57344 – 61439
13	53248 – 57343
12	49152 – 53247
11	45056 – 49151
10	40960 – 45055
9	36864 – 40959
8	32768 – 36863
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(a)

Page frame	Bottom 32K of main memory Physical addresses
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(b)

Figure 6-3. (a) The first 64K of virtual address space divided into 16 pages, with each page being 4K. (b) A 32K main memory divided up into eight page frames of 4K each.

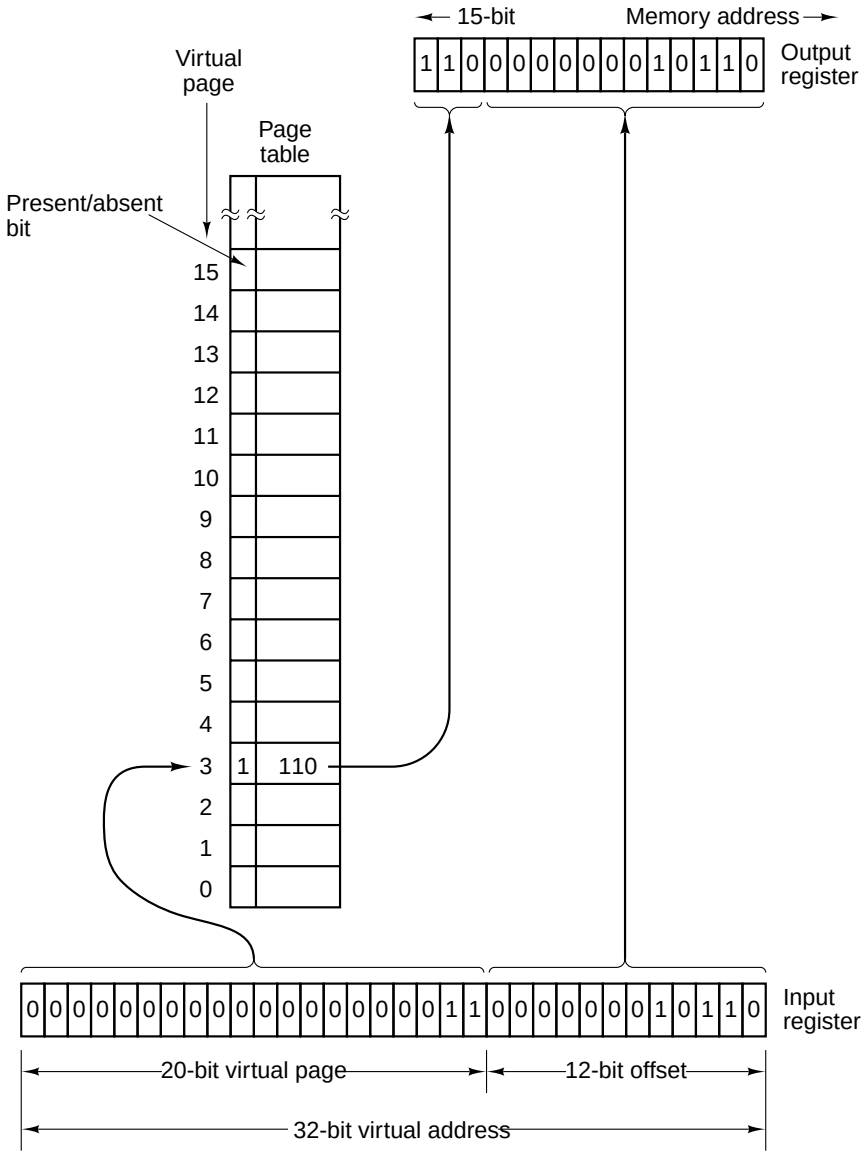


Figure 6-4. Formation of a main memory address from a virtual address.

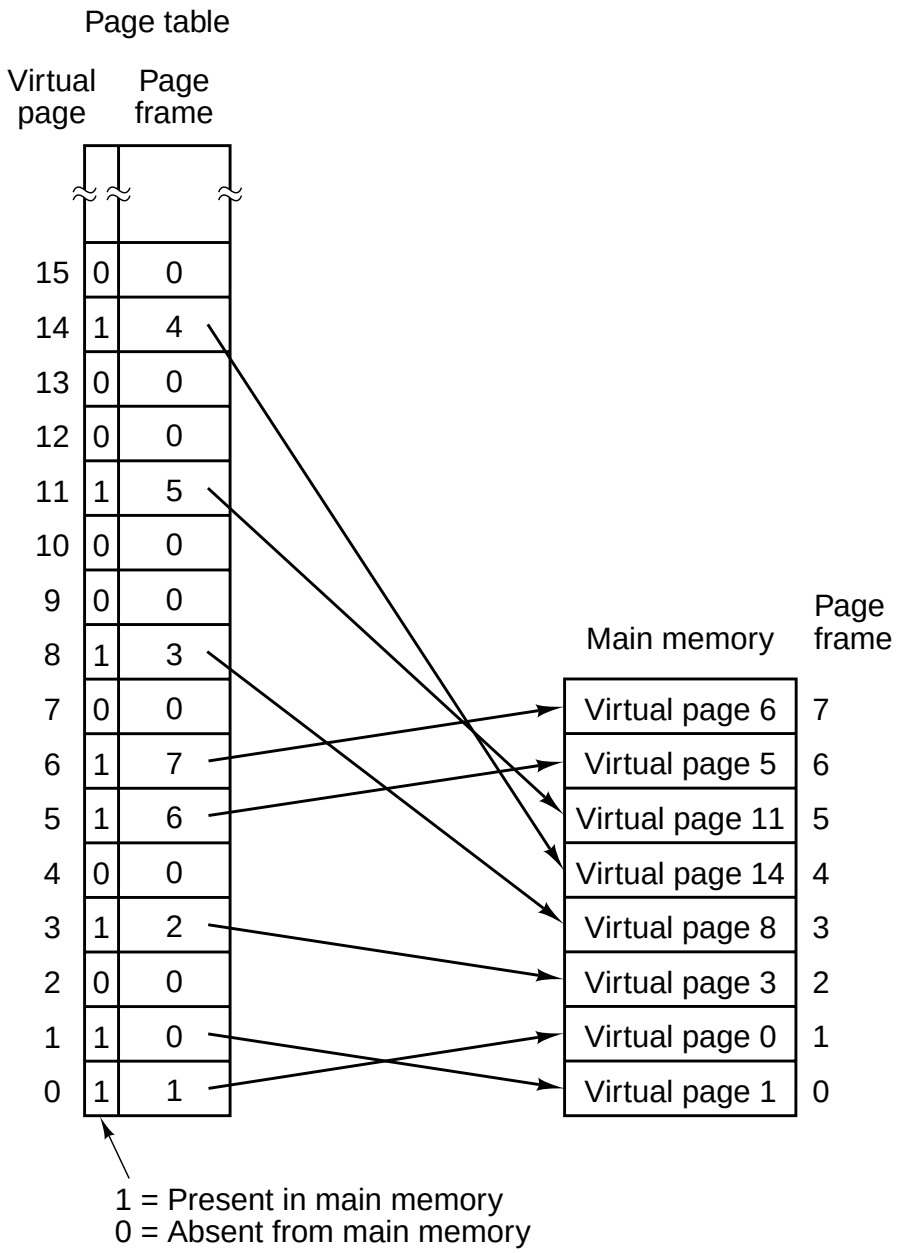


Figure 6-5. A possible mapping of the first 16 virtual pages onto a main memory with eight page frames.

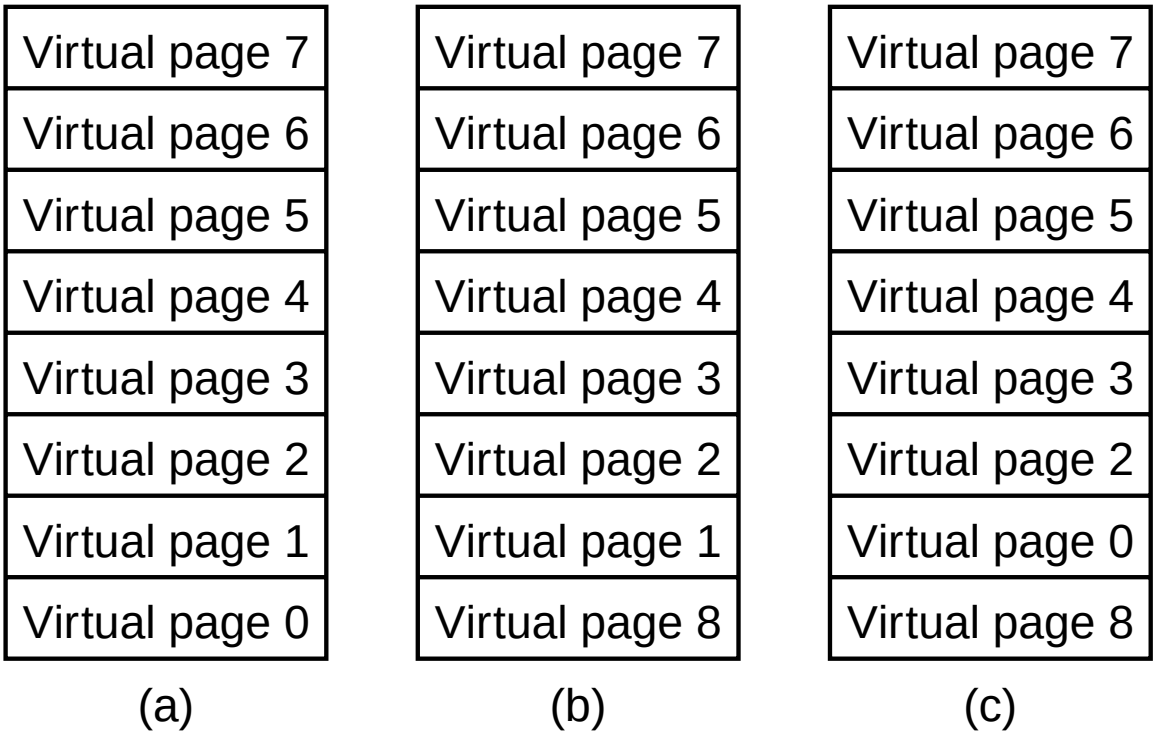


Figure 6-6. Failure of the LRU algorithm.

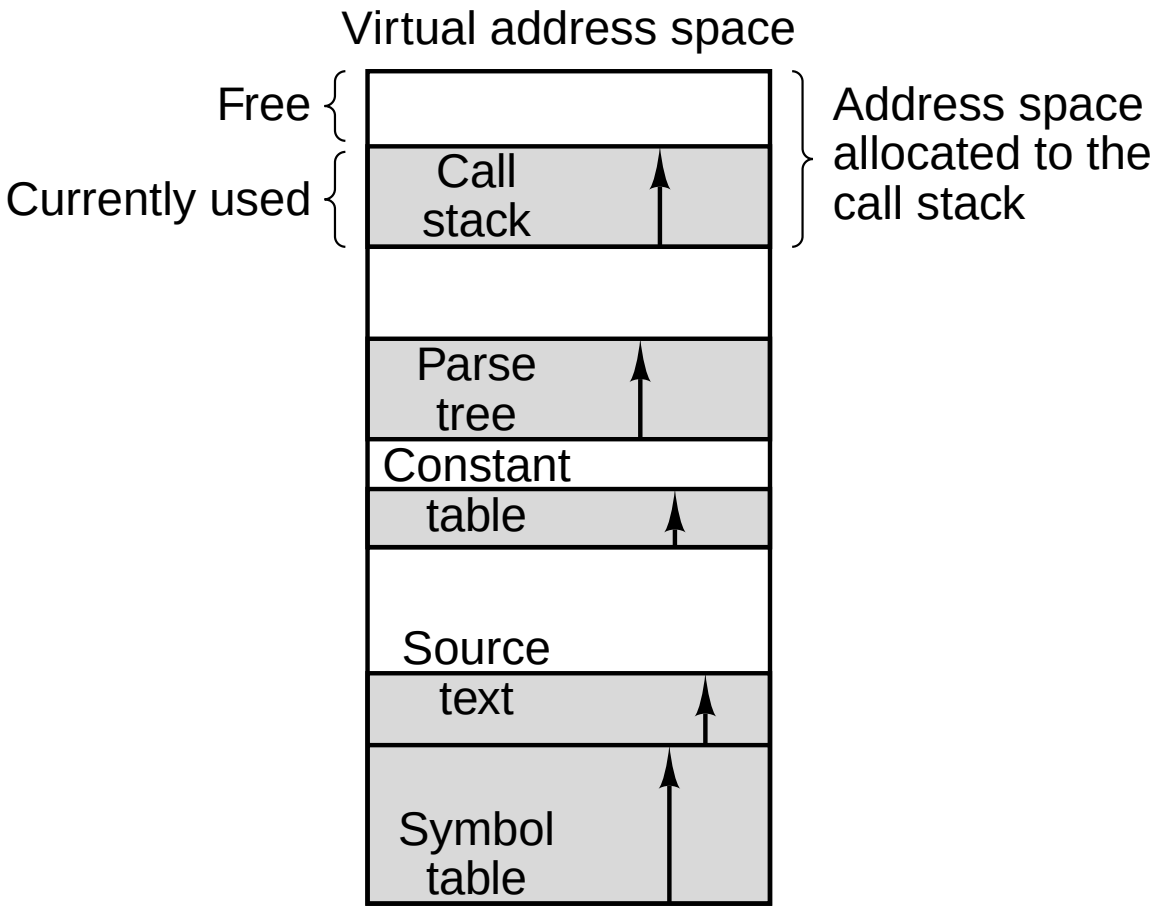


Figure 6-7. In a one-dimensional address space with growing tables, one table may bump into another.

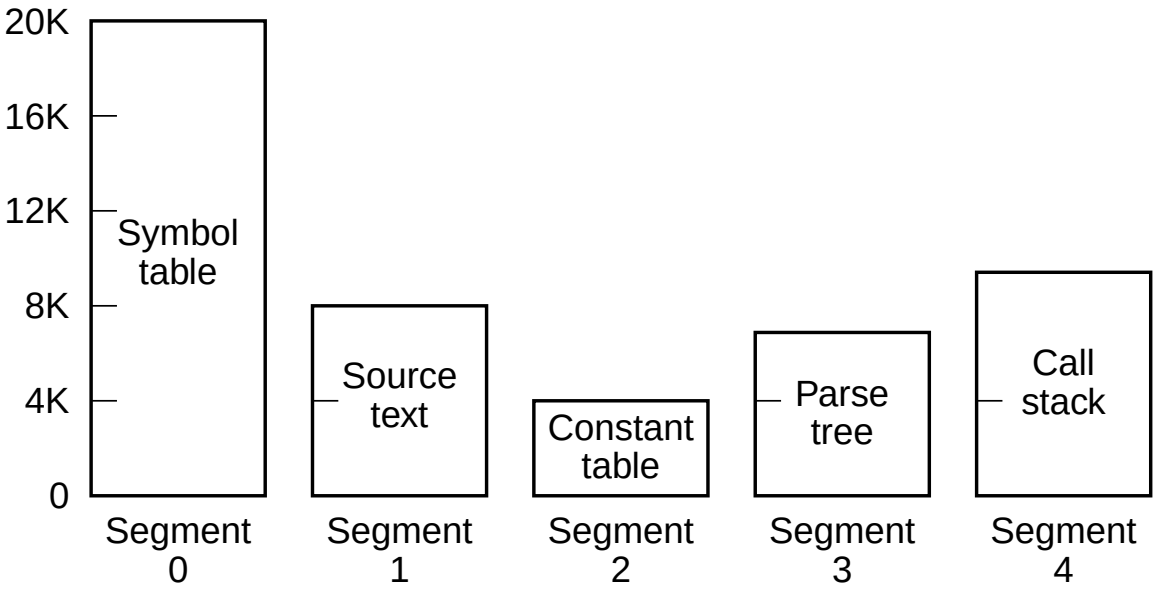


Figure 6-8. A segmented memory allows each table to grow or shrink independently of the other tables.

Consideration	Paging	Segmentation
Need the programmer be aware of it?	No	Yes
How many linear addresses spaces are there?	1	Many
Can virtual address space exceed memory size?	Yes	Yes
Can variable-sized tables be handled easily?	No	Yes
Why was the technique invented?	To simulate large memories	To provide multiple address spaces

Figure 6-9. Comparison of paging and segmentation.

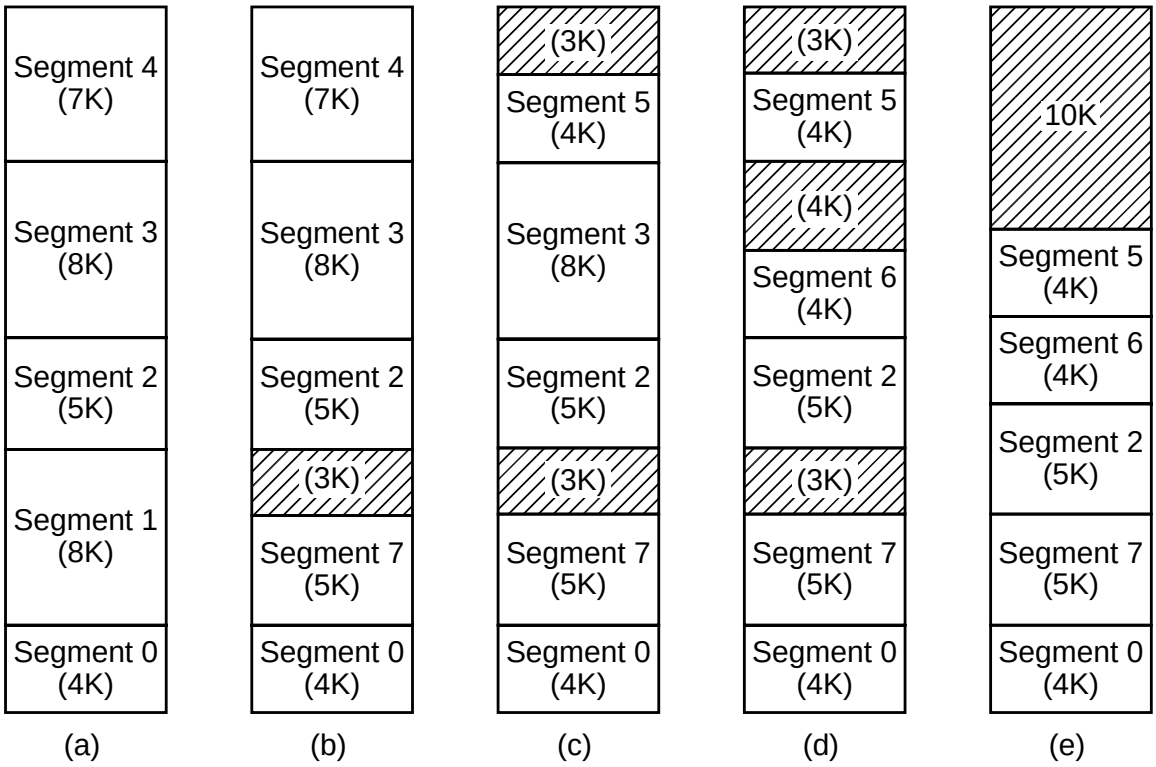


Figure 6-10. (a)-(d) Development of external fragmentation (e) Removal of the external fragmentation by compaction.

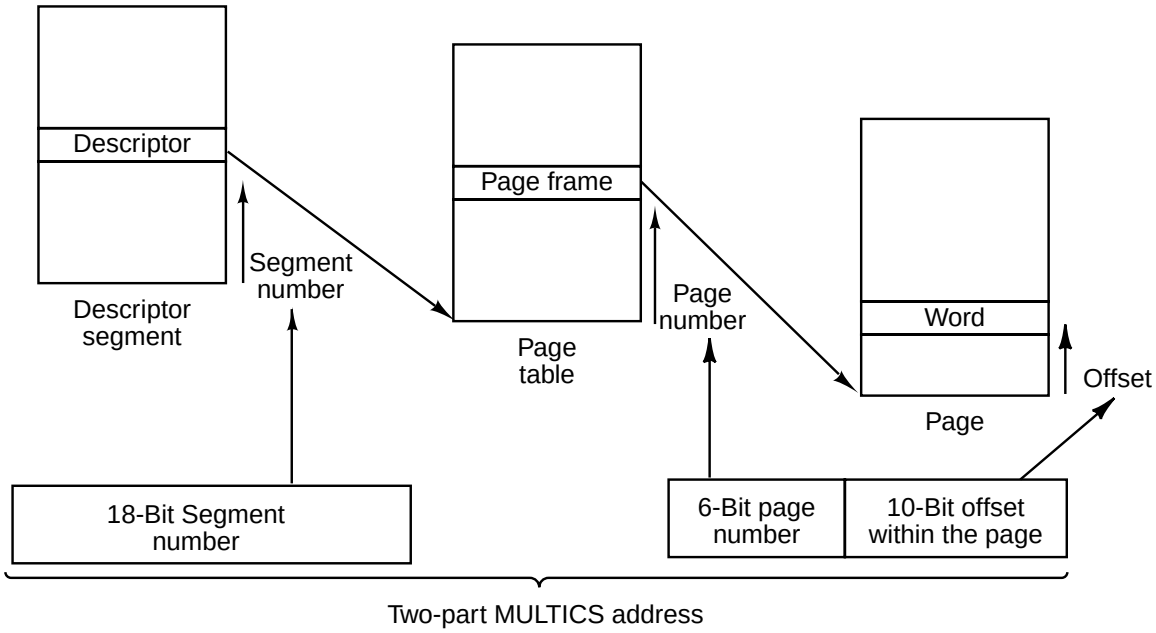


Figure 6-11. Conversion of a two-part MULTICS address into a main memory address.

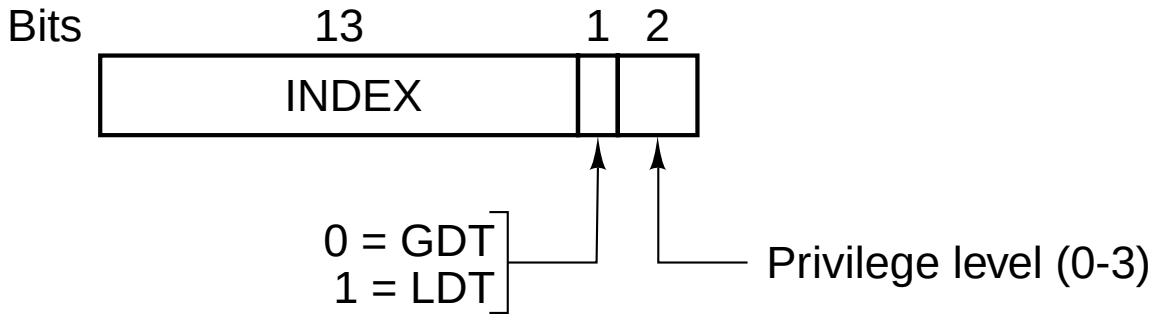


Figure 6-12. A Pentium II selector.

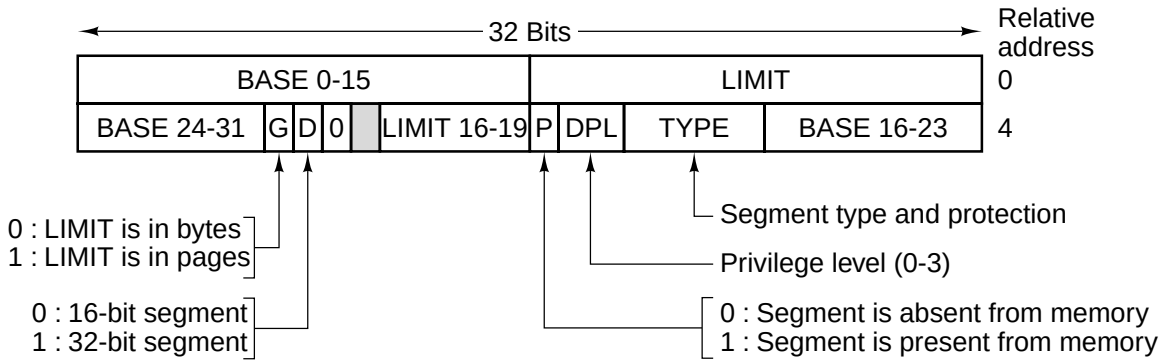


Figure 6-13. A Pentium II code segment descriptor. Data segments differ slightly.

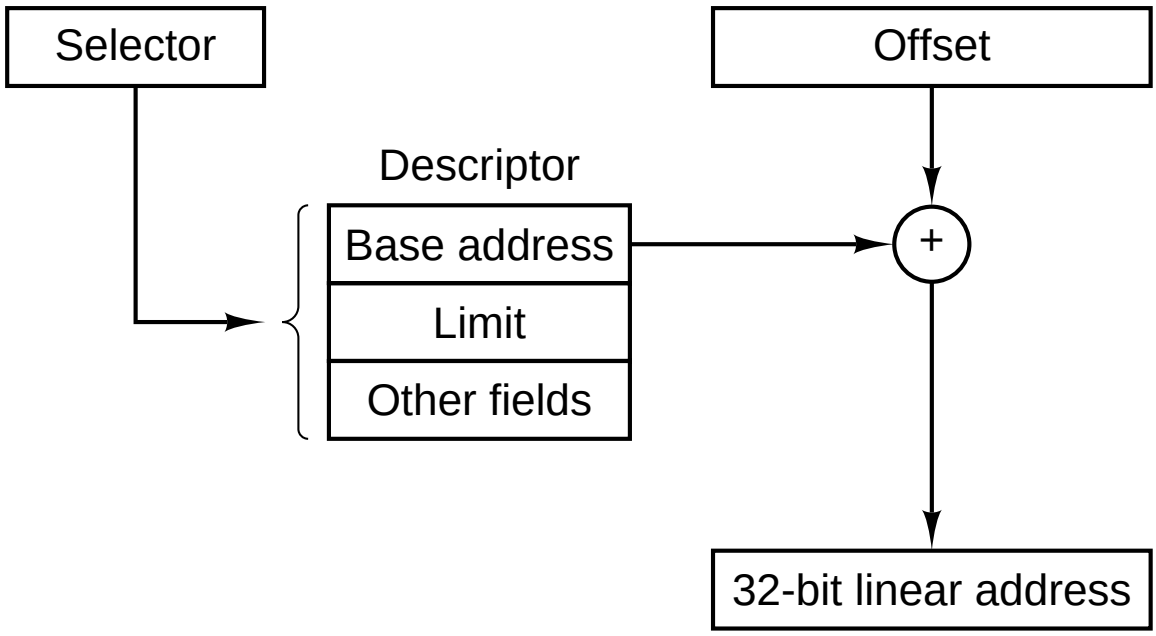


Figure 6-14. Conversion of a (selector, offset) pair to a linear address.

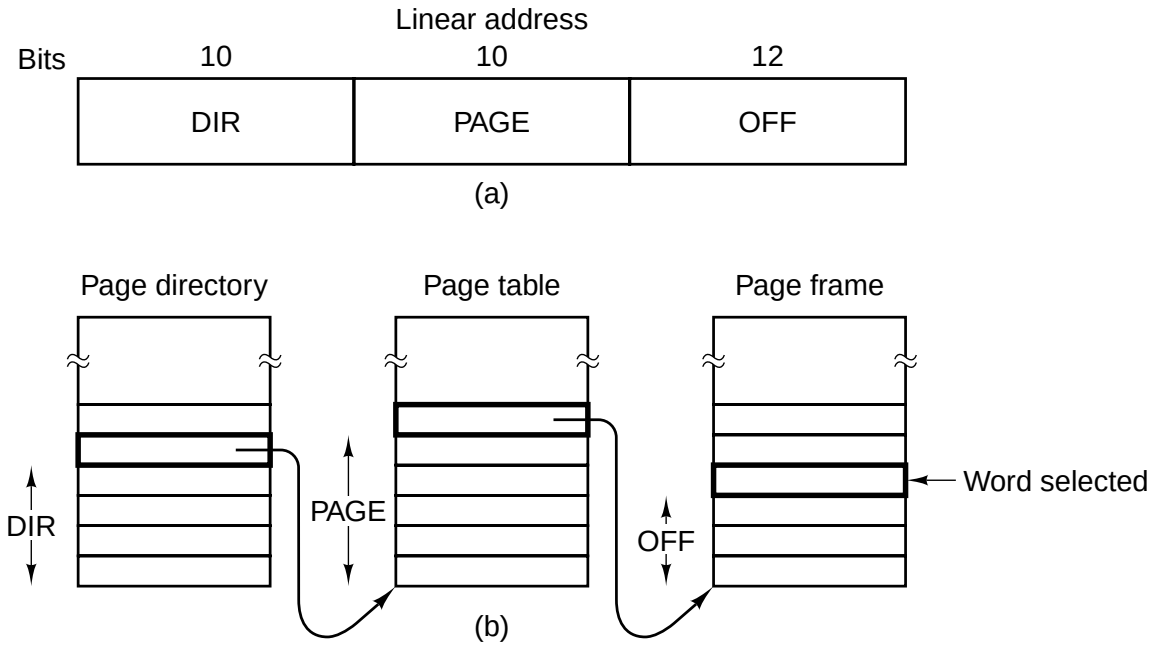


Figure 6-15. Mapping of a linear address onto a physical address.

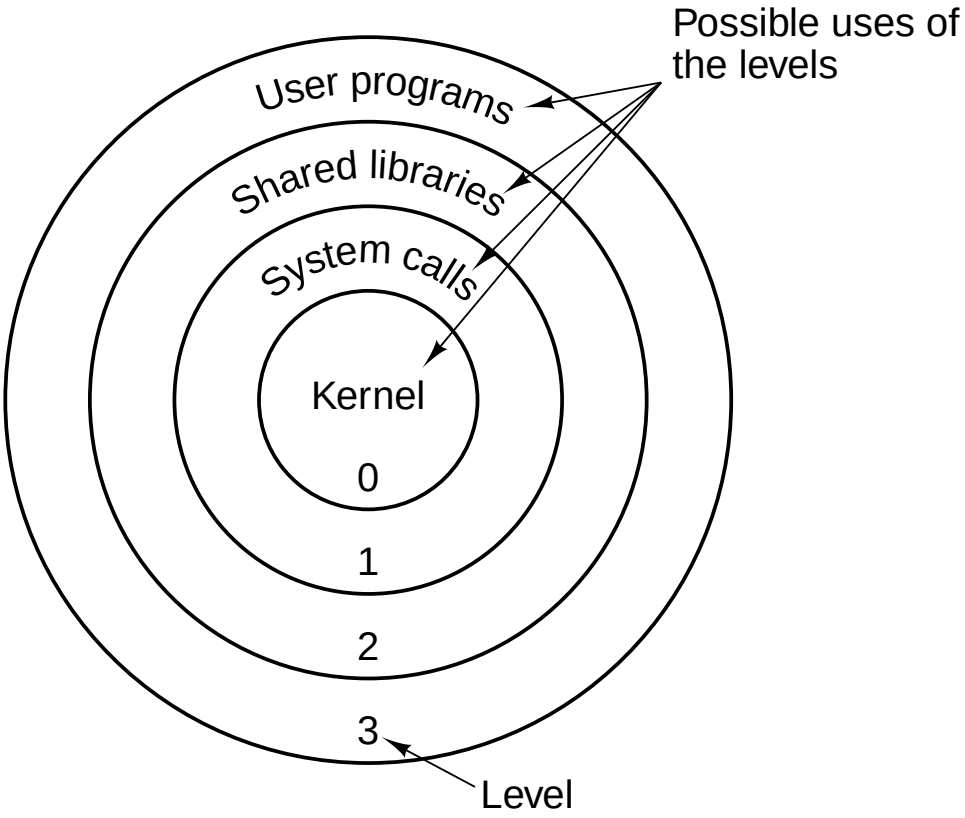


Figure 6-16. Protection on the Pentium II.

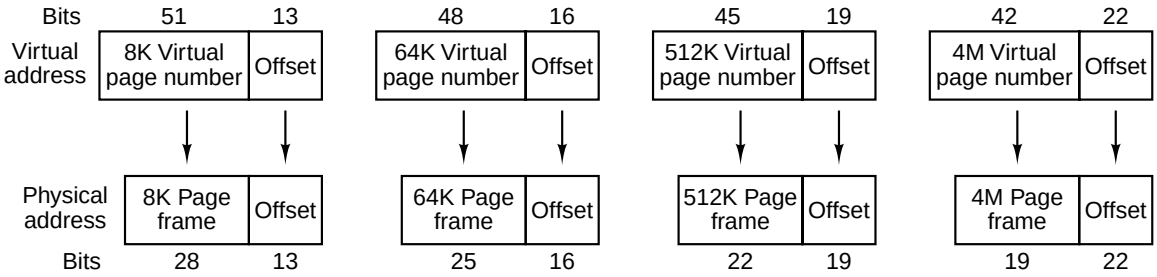


Figure 6-17. Virtual to physical mappings on the UltraSPARC.

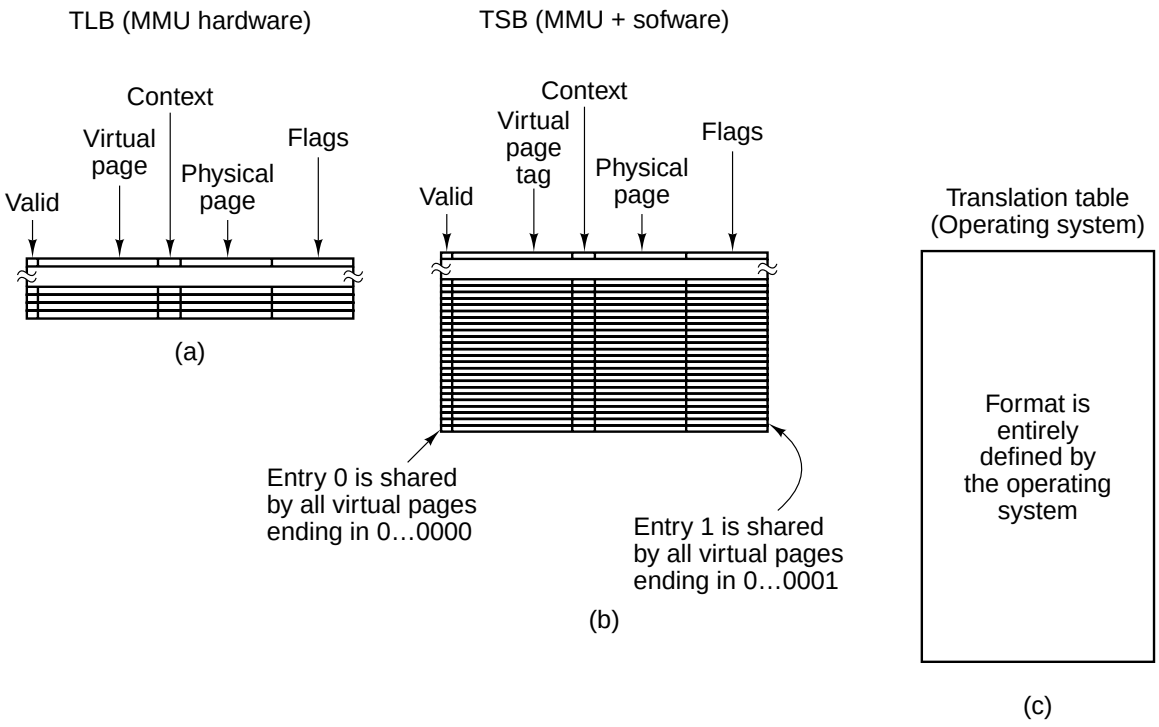


Figure 6-18. Data structures used in translating virtual addresses on the UltraSPARC. (a) TLB. (b) TSB. (c) Translation table.

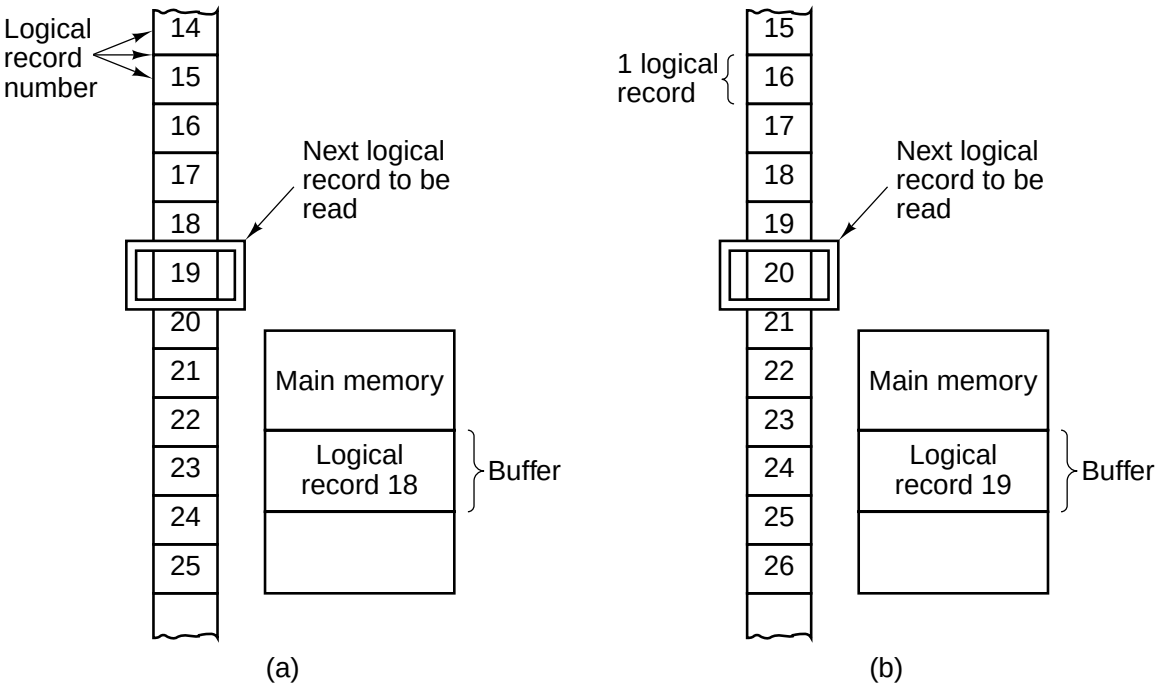


Figure 6-19. Reading a file consisting of logical records. (a) Before reading record 19. (b) After reading record 19.

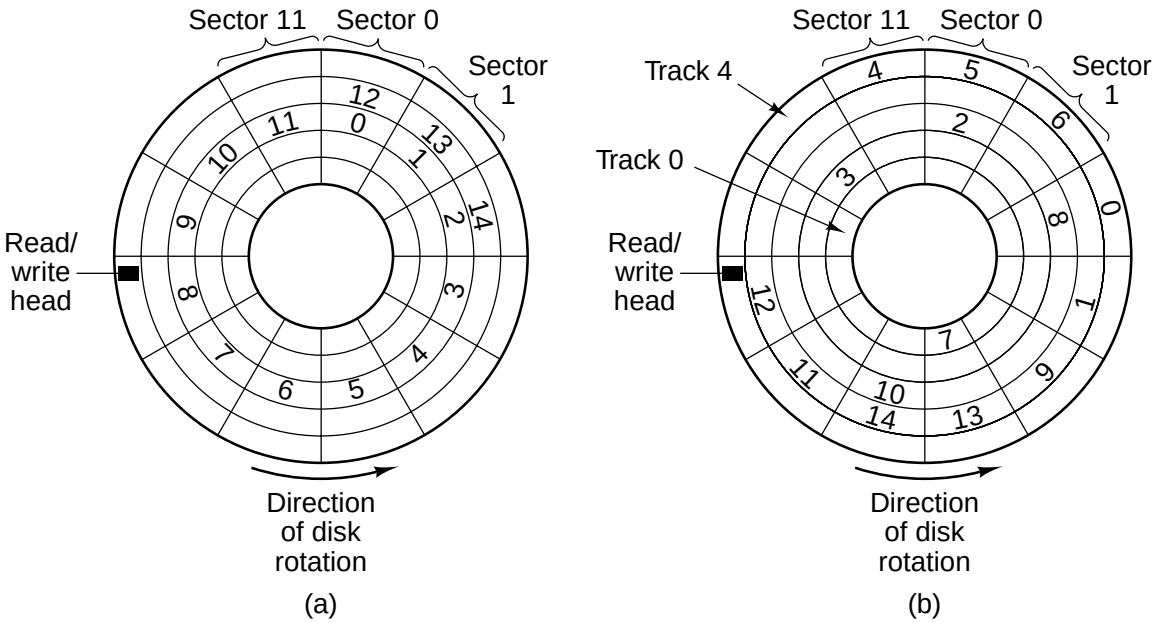


Figure 6-20. Disk allocation strategies. (a) A file in consecutive sectors. (b) A file not in consecutive sectors.

Track	Sector	Number of sectors in hole
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

(a)

	Sector											
Track	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

(b)

Figure 6-21. Two ways of keeping track of available sectors. (a) A free list. (b) A bit map.

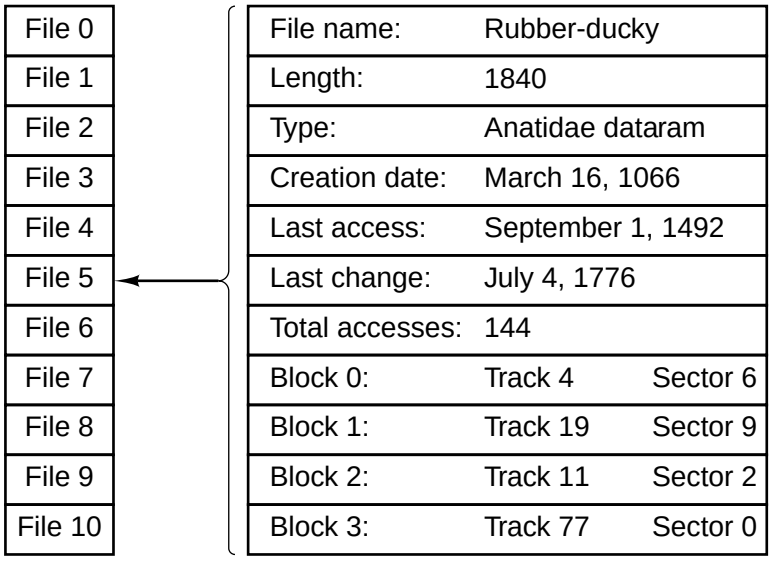


Figure 6-22. (a) A user file directory. (b) The contents of a typical entry in a file directory.

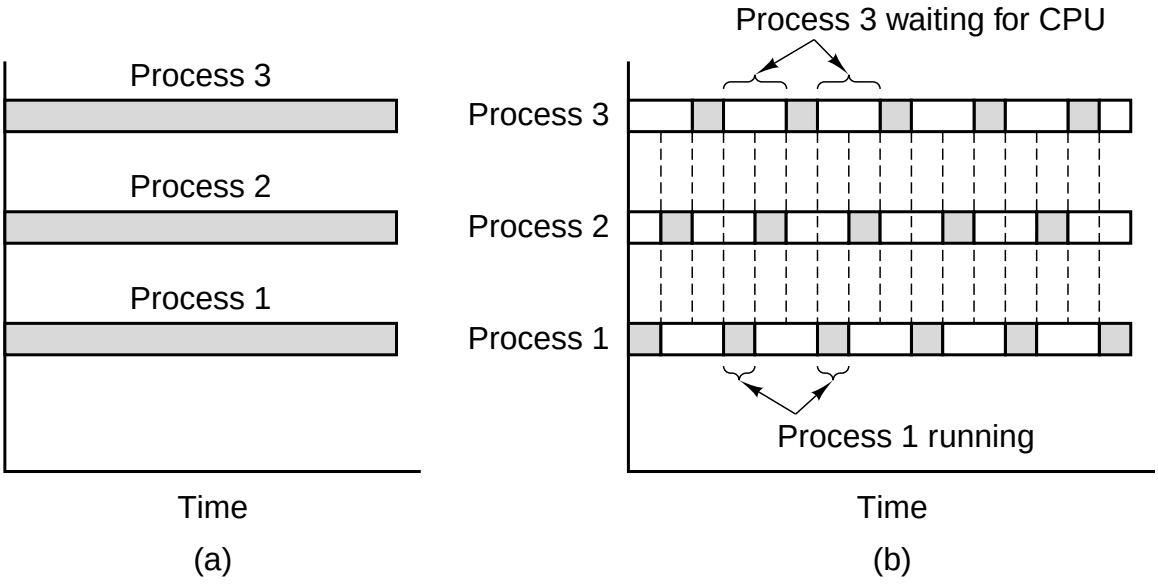


Figure 6-23. (a) True parallel processing with multiple CPUs. (b) Parallel processing simulated by switching one CPU among three processes.

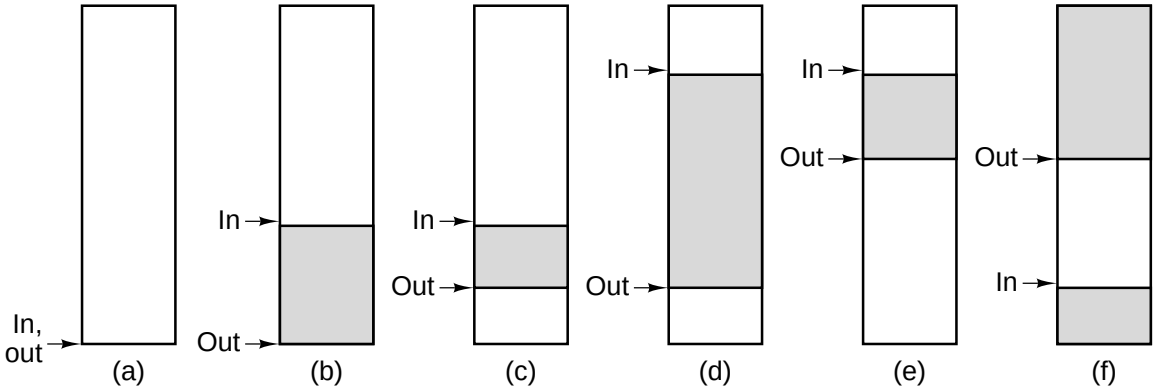


Figure 6-24. Use of a circular buffer.

```
public class m {
   final public static int BUF_SIZE = 100; // buffer runs from 0 to 99
   final public static long MAX_PRIME = 100000000000L; // stop here
   public static int in = 0, out = 0;          // pointers to the data
   public static long buffer[ ] = new long[BUF_SIZE];// primes stored here
   public static producer p;                   // name of the producer
   public static consumer c;                   // name of the consumer

   public static void main(String args[ ]) { // main class
      p = new producer( );                     // create the producer
      c = new consumer( );                     // create the consumer
      p.start( );                              // start the producer
      c.start( );                              // start the consumer
   }

   // This is a utility function for circularly incrementing in and out
   public static int next(int k) {if (k < BUF_SIZE – 1) return(k+1); else return(0);}
}

class producer extends Thread {          // producer class
   public void run( ) {                  // producer code
      long prime = 2;                    // scratch variable

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);      // statement P1
         if (m.next(m.in) == m.out) suspend( );       // statement P2
         m.buffer[m.in] = prime;         // statement P3
         m.in = m.next(m.in);            // statement P4
         if (m.next(m.out) == m.in) m.c.resume( );  // statement P5
      }
   }

   private long next_prime(long prime){ ... } // function that computes next prime
}

class consumer extends Thread {          // consumer class
   public void run( ) {                  // consumer code
      long emirp = 2;                    // scratch variable

      while (emirp < m.MAX_PRIME) {
         if (m.in == m.out) suspend( );  // statement C1
         emirp = m.buffer[m.out];        // statement C2
         m.out = m.next(m.out);          // statement C3
         if (m.out == m.next(m.next(m.in))) m.p.resume( );// statement C4
         System.out.println(emirp);      // statement C5
      }
   }
}
```

Figure 6-25. Parallel processing with a fatal race condition.

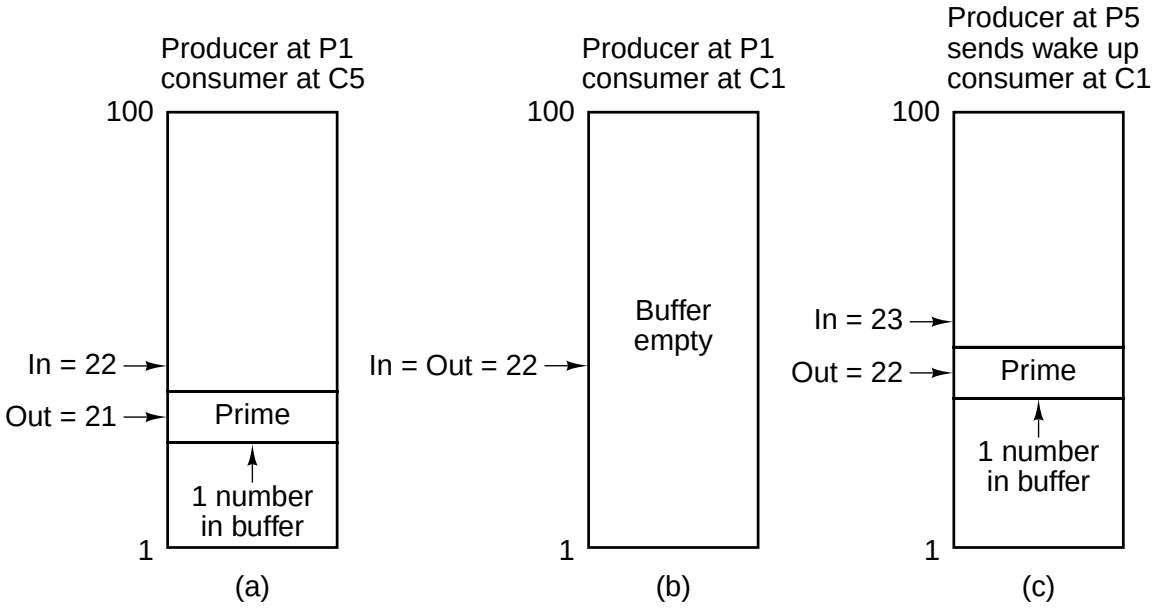


Figure 6-26. Failure of the producer-consumer communication mechanism.

Instr	Semaphore = 0	Semaphore > 0
Up	Semaphore=semaphore+1; if the other process was halted attempting to complete a down instruction on this semaphore, it may now complete the down and continue running	Semaphore=semaphore+1
Down	Process halts until the other process ups this semaphore	Semaphore=semaphore–1

Figure 6-27. The effect of a semaphore operation.

```
public class m {
   final public static int BUF_SIZE = 100; // buffer runs from 0 to 99
   final public static long MAX_PRIME = 100000000000L; // stop here
   public static int in = 0, out = 0;          // pointers to the data
   public static long buffer[ ] = new long[BUF_SIZE];// primes stored here
   public static producer p;                   // name of the producer
   public static consumer c;                   // name of the consumer
   public static int filled = 0, available = 100;        // semaphores

   public static void main(String args[ ]) { // main class
      p = new producer( );                     // create the producer
      c = new consumer( );                     // create the consumer
      p.start( );                              // start the producer
      c.start( );                              // start the consumer
   }

   // This is a utility function for circularly incrementing in and out
   public static int next(int k) {if (k < BUF_SIZE – 1) return(k+1); else return(0);}
}

class producer extends Thread {          // producer class
   native void up(int s); native void down(int s);     // methods on semaphores
   public void run( ) {                     // producer code
      long prime = 2;                       // scratch variable

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);         // statement P1
         down(m.available);                 // statement P2
         m.buffer[m.in] = prime;            // statement P3
         m.in = m.next(m.in);               // statement P4
         up(m.filled);                      // statement P5
      }
   }

   private long next_prime(long prime){ ... } // function that computes next prime
}

class consumer extends Thread {          // consumer class
   native void up(int s); native void down(int s);     // methods on semaphores
   public void run( ) {                     // consumer code
      long emirp = 2;                       // scratch variable

      while (emirp < m.MAX_PRIME) {
         down(m.filled);                    // statement C1
         emirp = m.buffer[m.out];           // statement C2
         m.out = m.next(m.out);             // statement C3
         up(m.available);                   // statement C4
         System.out.println(emirp);         // statement C5
      }
   }
}
```

Figure 6-28. Parallel processing using semaphores.

Category	Some examples
File management	Open, read, write, close, and lock files
Directory management	Create and delete directories; move files around
Process management	Spawn, terminate, trace, and signal processes
Memory management	Share memory among processes; protect pages
Getting/setting parameters	Get user, group, process ID; set priority
Dates and times	Set file access times; use interval timer; profile execution
Networking	Establish/accept connection; send/receive message
Miscellaneous	Enable accounting; manipulate disk quotas; reboot the system

Figure 6-29. A rough breakdown of the UNIX system calls.

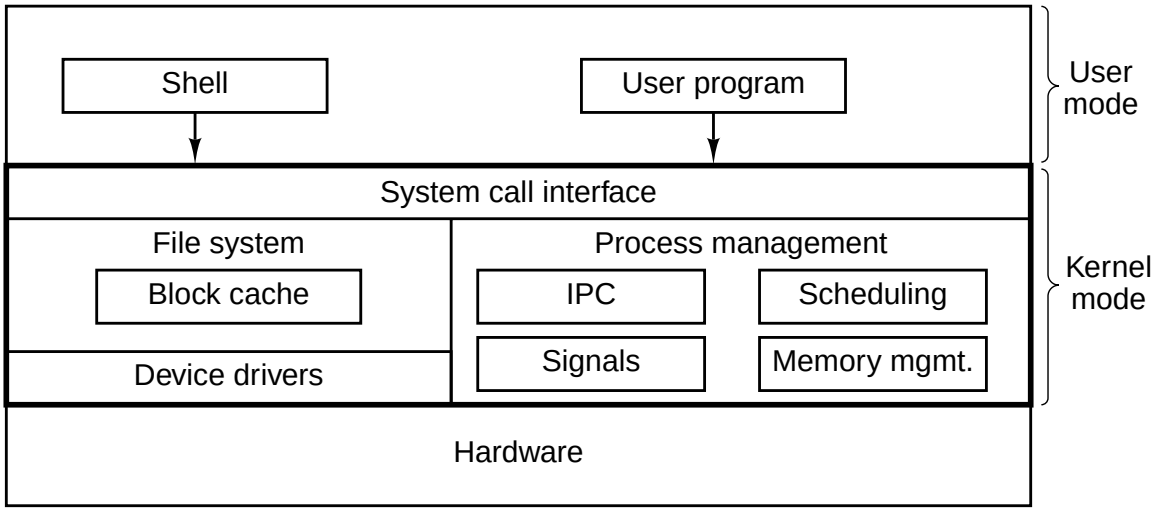


Figure 6-30. The structure of a typical UNIX system.

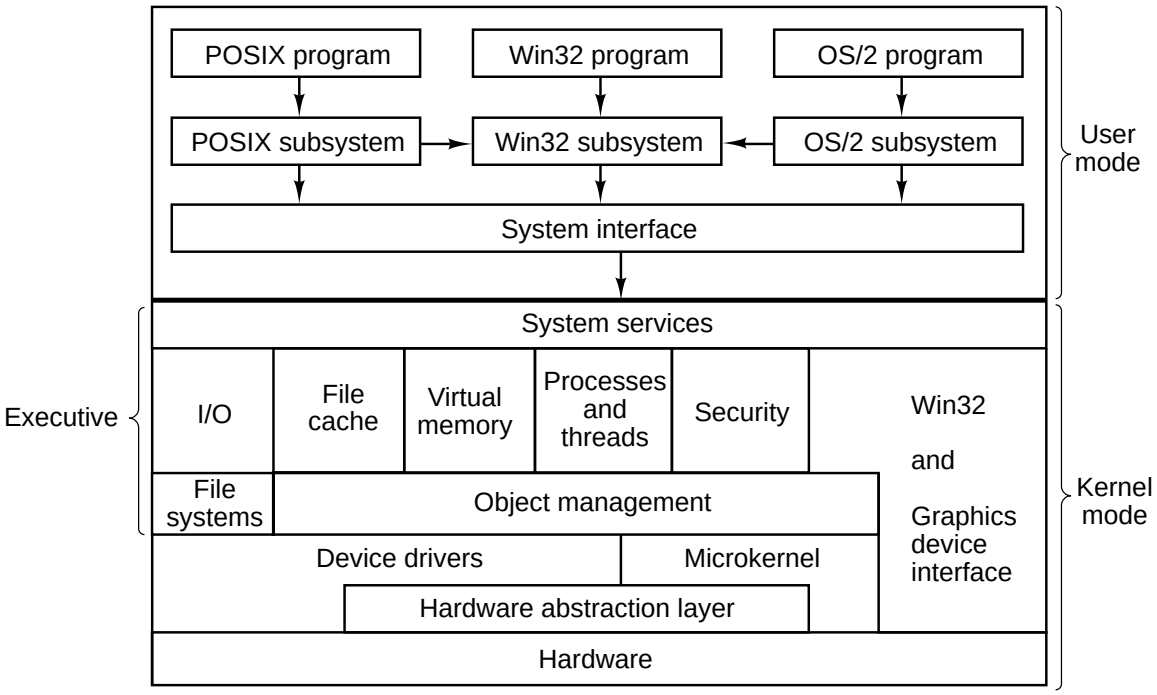


Figure 6-31. The structure of Windows NT.

Item	Windows 95/98	NT 5.0
Win32 API?	Yes	Yes
Full 32-bit system?	No	Yes
Security?	No	Yes
Protected file mappings?	No	Yes
Sep. addr space for each MS-DOS program?	No	Yes
Plug and play?	Yes	Yes
Unicode?	No	Yes
Runs on	Intel 80x86	80x86, Alpha
Multiprocessor support?	No	Yes
Re-entrant code inside OS?	No	Yes
Some critical OS data writable by user?	Yes	No

Figure 6-32. Some differences between versions of Windows.

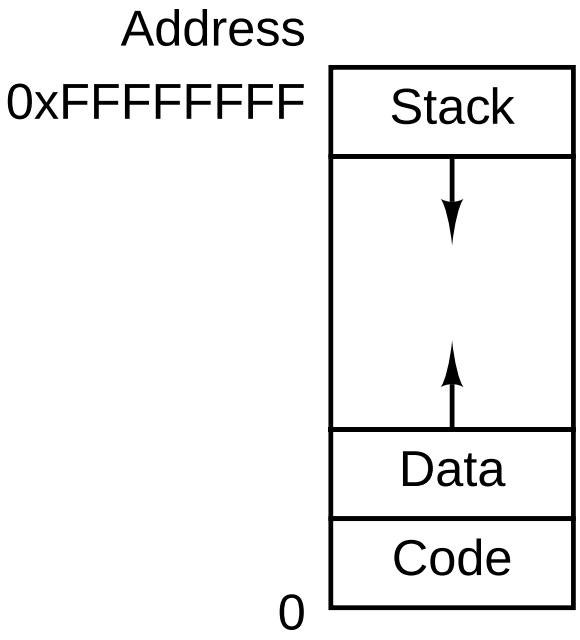


Figure 6-33. The address space of a single UNIX process.

API function	Meaning
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it)
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space
UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file mapping object

Figure 6-34. The principal API functions for managing virtual memory in Windows NT.

System call	Meaning
creat(name, mode)	Create a file; *mode* specifies the protection mode
unlink(name)	Delete a file (assuming that there is only 1 link to it)
open(name, mode)	Open or create a file and return a file descriptor
close(fd)	Close a file
read(fd, buffer, count)	Read *count* bytes into *buffer*
write(fd, buffer, count)	Write *count* bytes from *buffer*
lseek(fd, offset, w)	Move the file pointer as required by *offset* and *w*
stat(name, buffer)	Return information about a file
chmod(name, mode)	Change the protection mode of a file
fcntl(fd, cmd, ...)	Do various control operations such as locking (part of) a file

Figure 6-35. The principal UNIX file system calls.

```
// Open the file descriptors
infd = open(″data″, 0);
outfd = creat(″newf″, ProtectionBits);

// Copy loop
do {
    count = read(infd, buffer, bytes);
    if (count > 0) write(outfd, buffer, count);
} while (count > 0);

// Close the files
close(infd);
close(outfd);
```

Figure 6-36. A program fragment for copying a file using the UNIX system calls. This fragment is in C because Java hides the low-level system calls and we are trying to expose them.

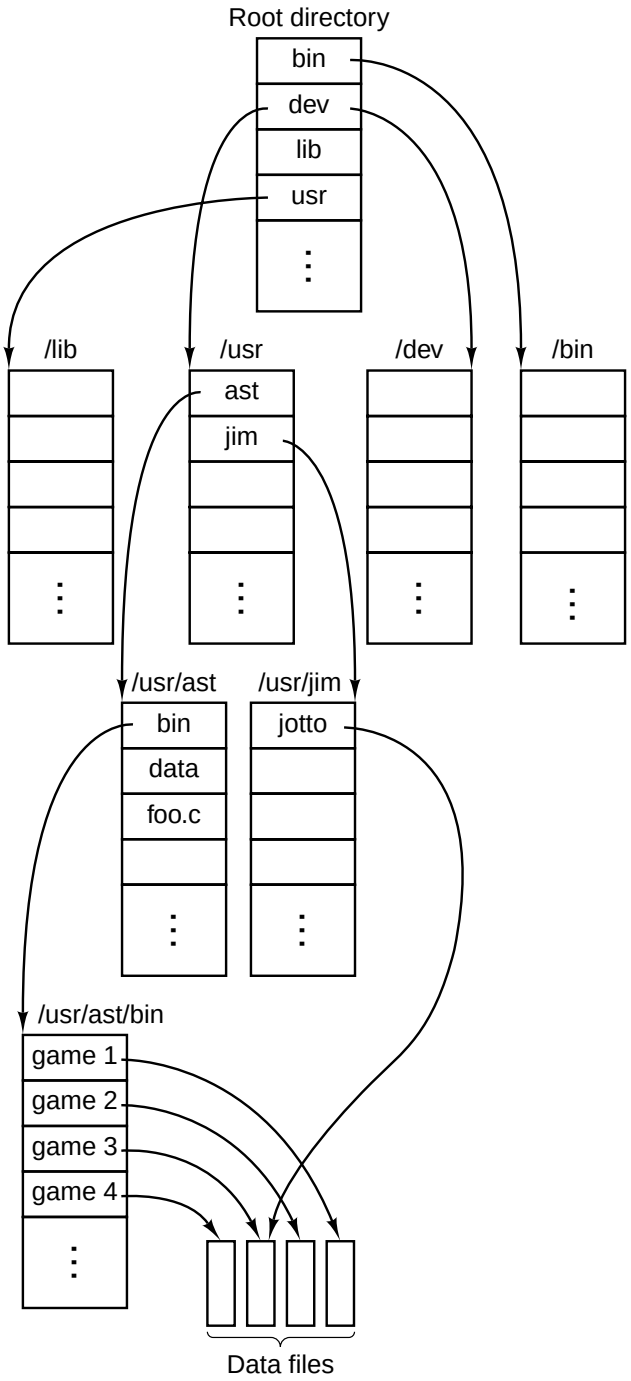


Figure 6-37. Part of a typical UNIX directory system.

System call	Meaning
mkdir(name, mode)	Create a new directory
rmdir(name)	Delete an empty directory
opendir(name)	Open a directory for reading
readdir(dirpointer)	Read the next entry in a directory
closedir(dirpointer)	Close a directory
chdir(dirname)	Change working directory to *dirname*
link(name1, name2)	Create a directory entry *name2* pointing to *name1*
unlink(name)	Remove *name* from its directory

Figure 6-38. The principal UNIX directory management calls.

API function	UNIX	Meaning
CreateFile	open	Create a file or open an existing file; return a handle
DeleteFile	unlink	Destroy an existing file
CloseHandle	close	Close a file
ReadFile	read	Read data from a file
WriteFile	write	Write data to a file
SetFilePointer	lseek	Set the file pointer to a specific place in the file
GetFileAttributes	stat	Return the file properties
LockFile	fcntl	Lock a region of the file to provide mutual exclusion
UnlockFile	fcntl	Unlock a previously locked region of the file

Figure 6-39. The principal Win32 API functions for file I/O. The second column gives the nearest UNIX equivalent.

```
// Open files for input and output.
inhandle = CreateFile(″data″, GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile(″newf″, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);

// Copy the file.
do {
        s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
        if (s > 0 && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
while (s > 0 && count > 0);

// Close the files.
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Figure 6-40. A program fragment for copying a file using the Windows NT API functions. This fragment is in C because Java hides the low-level system calls and we are trying to expose them.

API function	UNIX	Meaning
CreateDirectory	mkdir	Create a new directory
RemoveDirectory	rmdir	Remove an empty directory
FindFirstFile	opendir	Initialize to start reading the entries in a directory
FindNextFile	readdir	Read the next directory entry
MoveFile		Move a file from one directory to another
SetCurrentDirectory	chdir	Change the current working directory

Figure 6-41. The principal Win32 API functions for directory management. The second column gives the nearest UNIX equivalent, when one exists.

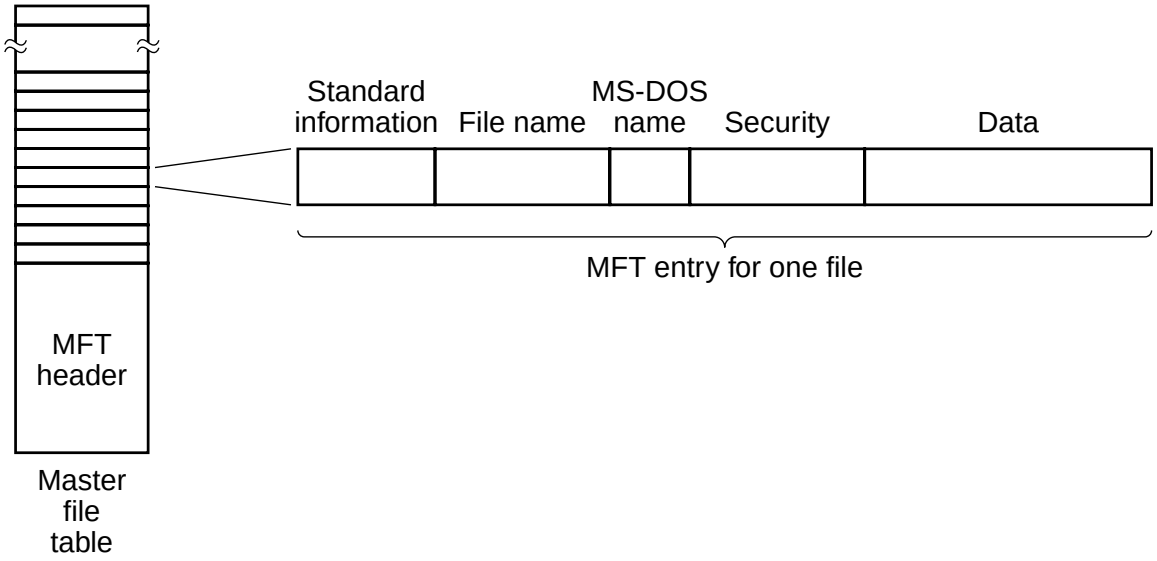


Figure 6-42. The Windows NT master file table.

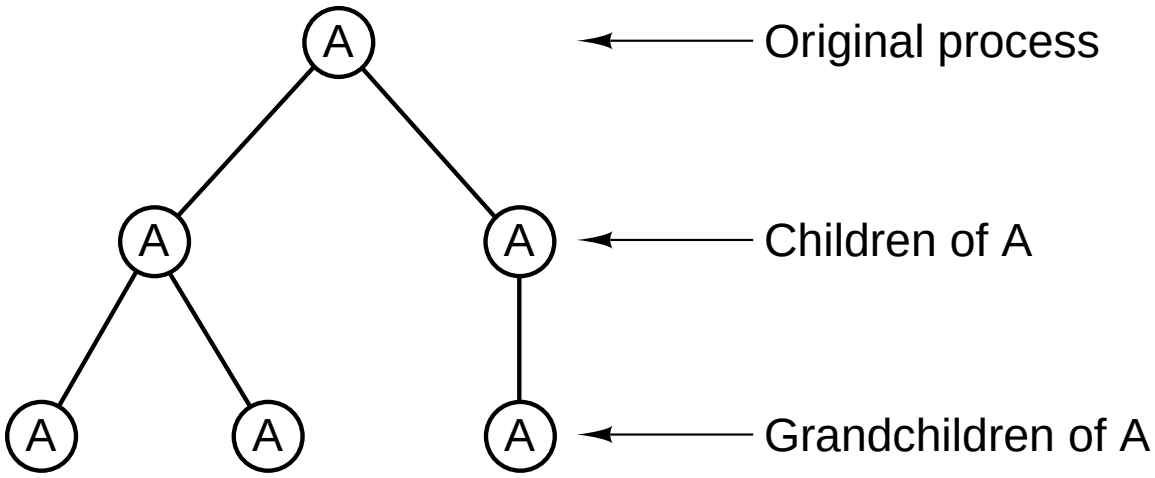


Figure 6-43. A process tree in UNIX.

Thread call	Meaning
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Figure 6-44. The principal POSIX thread calls.

6.4 EXAMPLE OPERATING SYSTEMS

In this section we will continue discussing our example systems, the Pentium II and the UltraSPARC. For each one we will look at an operating system used on that processor. For the Pentium II we will use Windows NT (called NT for short below); for the UltraSPARC II we will use UNIX. Since UNIX is simpler and in many ways more elegant, we will begin with it. Also, UNIX was designed and implemented first and had a major influence on NT, so this order makes more sense than the reverse.

6.4.1 Introduction

In this section we will give a brief introduction to our two example operating systems, UNIX and NT, focusing on the history, structure, and system calls.

UNIX

UNIX was developed at Bell Labs in the early 1970s. The first version was written by Ken Thompson in assembler for the PDP-7 minicomputer. This was soon followed by a version for the PDP-11, written in a new language called C that was devised and implemented by Dennis Ritchie. In 1974, Ritchie and Thompson published a landmark paper about UNIX (Ritchie and Thompson, 1974). For the work described in this paper they were later given the prestigious ACM Turing Award (Ritchie, 1984; Thompson, 1984). The publication of this paper stimulated many universities to ask Bell Labs for a copy of UNIX. Since Bell Labs' parent company, AT&T, was a regulated monopoly at the time and was not permitted to be in the computer business, it had no objection to licensing UNIX to universities for a modest fee.

In one of those coincidences that often shape history, the PDP-11 was the computer of choice at nearly all university computer science departments, and the operating systems that came with the PDP-11 were widely regarded as being dreadful by professors and students alike. UNIX quickly filled the void, not in the least because it was supplied with the complete source code, so people could, and did, tinker with it endlessly.

One of the many universities that acquired UNIX early on was the University of California at Berkeley. Because the complete source code was available, Berkeley was able to modify the system substantially. Foremost among the changes was a port to the VAX minicomputer and the addition of paged virtual memory, the extension of file names from 14 characters to 255 characters, and the inclusion of the TCP/IP networking protocol, which is now used on the Internet (largely due to the fact that it was in Berkeley UNIX).

While Berkeley was making all these changes, AT&T itself continued to develop UNIX, leading to System III in 1982 and then System V in 1984. By the

late 1980s, two different, and quite incompatible, versions of UNIX were in widespread use: Berkeley UNIX and System V. This split in the UNIX world, together with the fact that there were no standards for binary program formats, greatly inhibited the commercial success of UNIX because it was impossible for software vendors to write and package UNIX programs with the expectation that they would run on any UNIX system (as was routinely done with MS-DOS). After much bickering, a standard called **POSIX** (**Portable Operating System-IX**) was created by the IEEE Standards Board. POSIX is also known by its IEEE Standards number, P1003. It later became an International Standard.

The POSIX standard is divided into many parts, each covering a different area of UNIX. The first part, P1003.1, defines the system calls; the second part, P1003.2, defines the basic utility programs, and so on. The P1003.1 standard defines about 60 system calls that all conformant systems must support. These are the basic calls for reading and writing files, creating new processes, and so on. Nearly all UNIX systems now support the P1003.1 system calls. However many UNIX systems also support extra system calls, especially those defined by System V and/or those in Berkeley UNIX. Typically these add up to perhaps 100 system calls to the POSIX set. The operating system for the UltraSPARC II is based on System V and is called **Solaris**. It also supports many of the Berkeley system calls.

A rough breakdown of the Solaris system calls by category is given in Fig. 6-1. The file and directory management system calls are largest and the most important categories. Most of these come from P1003.1. A relatively large fraction of the others are derived from System V.

Category	Some examples
File management	Open, read, write, close, and lock files
Directory management	Create and delete directories; move files around
Process management	Spawn, terminate, trace, and signal processes
Memory management	Share memory among processes; protect pages
Getting/setting parameters	Get user, group, process ID; set priority
Dates and times	Set file access times; use interval timer; profile execution
Networking	Establish/accept connection; send/receive message
Miscellaneous	Enable accounting; manipulate disk quotas; reboot the system

Figure 6-1. A rough breakdown of the UNIX system calls.

One area that is largely due to Berkeley UNIX rather than System V is networking. Berkeley invented the concept of a **socket**, which is the endpoint of a network connection. The four-pin wall plugs to which telephones can be connected served as the model for this concept. It is possible for a UNIX process to create a socket, attach to it, and establish a connection to a socket on a distant

machine. Over this connection it can then exchange data in both directions, typically using the TCP/IP protocol. Since networking technology has been in UNIX for decades and is very stable and mature, a substantial fraction of the servers on the Internet run UNIX.

Since there are many implementations of UNIX, it is difficult to say much about the structure of the operating system since every one is somewhat different from all the others. However, in general, Fig. 6-2 applies to most of them. At the bottom, there is a layer of device drivers that shield the file system from the bare hardware. Originally, each device driver was written as an independent entity, separate from all the others. This arrangement led to a lot of duplicated effort, since many drivers must deal with flow control, error handling, priorities, separating data from control, and so on. This observation led Dennis Ritchie to develop a framework called **streams** for writing drivers in a modular way. With a stream, it is possible to establish a two-way connection from a user process to a hardware device and to insert one or more modules along the path. The user process pushes data into the stream, which then is processed or transformed by each module until it gets to the hardware. The inverse processing occurs for incoming data.

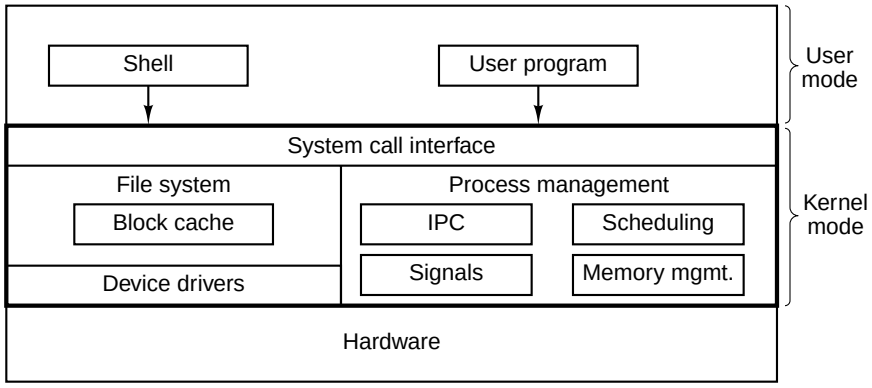


Figure 6-2. The structure of a typical UNIX system.

On top of the device drivers comes the file system. It manages file names, directories, disk block allocation, protection, and much more. Part of the file system is a **block cache**, for holding the blocks most recently read in from disk, in case they are needed again soon. A variety of file systems have been used over the years, including the Berkeley fast file system (McKusick et al., 1984), and log-structured file systems (Rosenblum and Ousterhout, 1991; and Seltzer et al., 1993).

The other part of the UNIX kernel is the process management portion. Among its various other functions, it handles IPC (InterProcess Communication), which allows processes to communicate with one another and synchronize to avoid race conditions. A variety of mechanisms are provided. The process management

code also handles process scheduling, which is based on priorities. Signals, which are a form of (asynchronous) software interrupt, are also managed here. Finally, memory management is done here as well. Most UNIX systems support demand-paged virtual memory, sometimes with a few extra features, such as the ability of multiple processes to share common regions of address space.

From its inception, UNIX has tried to be a small system, in order to enhance reliability and performance. The first versions of UNIX were entirely text-based, using terminals that could display 24 or 25 lines of 80 ASCII characters. The user interface was handled by a user-level program called the **shell**, which offered a command line interface. Since the shell was not part of the kernel, adding new shells to UNIX was easy, and over time a number of increasingly sophisticated ones were invented.

Later on, when graphics terminals came into existence, a windowing system for UNIX, called **X Windows**, was developed at M.I.T. Still later, a full-fledged **GUI** (**Graphical User Interface**), called **Motif**, was put on top of X Windows. In keeping with the UNIX philosophy of having a small kernel, nearly all the code of X Windows and Motif runs in user mode, outside the kernel.

Windows NT

When the original IBM PC was launched in 1981, it came equipped with a 16-bit real-mode, single-user, command-line oriented operating system called MS-DOS 1.0. This operating system consisted of 8 KB of memory resident code. Two years later, a much more powerful 24-KB system, MS-DOS 2.0, appeared. It contained a command line processor (shell), with a number of features borrowed from UNIX. When IBM released the 286-based PC/AT in 1984, it came equipped with MS-DOS 3.0, by now 36 KB. Over the years, MS-DOS continued to acquire new features, but it was still a command-line oriented system.

Inspired by the success of the Apple Macintosh, Microsoft decided to give MS-DOS a graphical user interface that it called **Windows**. The first three versions of Windows, culminating in Windows 3.x, were not true operating systems, but graphical user interfaces on top of MS-DOS, which was still in control of the machine. All programs ran in the same address space and a bug in any one of them could bring the whole system to a grinding halt.

The release of Windows 95 in 1995 still did not eliminate MS-DOS, although it introduced a new version, 7.0. Together, Windows 95 and MS-DOS 7.0 contained most of the features of a full-blown operating system, including virtual memory, process management, and multiprogramming. However, Windows 95 was not a full 32-bit program. It contained large chunks of old 16-bit code (as well as some 32-bit code) and still used the MS-DOS file system, with nearly all its limitations. The only major change to the file system was the addition of long file names in place of the 8 + 3 character file names allowed in MS-DOS.

Even with the release of Windows 98 in 1998, MS-DOS was still there (now

called version 7.1) and running 16-bit code. Although a bit more functionality migrated from the MS-DOS part to the Windows part, and a disk layout suitable for larger disks was now standard, under the hood, Windows 98 was not very different from Windows 95. The main difference was the user interface, which integrated the desktop, the Internet, and television more closely. It was precisely this integration that attracted the attention of the U.S. Dept. of Justice, which then sued Microsoft claiming that it was an illegal monopoly.

While all these developments were going on, Microsoft was also busy with a completely new 32-bit operating system being written from the ground up. This new system was called **Windows New Technology**, or **Windows NT**. It was initially hyped as the replacement for all other operating systems for Intel-based PCs, but it was somewhat slow to catch on and was later redirected to the upper end of the market, where it found a niche. It is gradually becoming more popular at the low end as well.

NT is sold in two versions: server and workstation. These two versions are nearly identical and are generated from the same source code. The server version is intended for machines that run as LAN-based file and print servers and has more elaborate management features than the workstation version, which is intended for desktop computing for a single user. The server version has a variant (enterprise) intended for large sites. The various versions are tuned differently, each one optimized for its expected environment. Other than these minor differences, all the versions are essentially the same. In fact, nearly all the executable files are identical for all versions. NT itself discovers which version it is by looking at a variable in an internal data structure (the registry). Users are forbidden by the license from changing this variable and thus converting the (inexpensive) workstation version into the (much more expensive) server or enterprise versions. We will not make any further distinction between these versions.

MS-DOS and all previous versions of Windows were single-user systems. NT, however, supports multiprogramming, so several users can work on the same machine at the same time. For example, a network server may have multiple users logged in simultaneously over a network, each accessing its own files in a protected way.

NT is a true 32-bit multiprogramming operating system. It supports multiple user processes, each of which has a full 32-bit demand-paged virtual address space. In addition, the system itself is written as 32-bit code everywhere.

One of NT's original improvements over Windows 95 was its modular structure. It consisted of a moderately small kernel that ran in kernel mode, plus a number of server processes that ran in user mode. User processes interacted with the server processes using the client-server model: a client sent a request message to a server, and the server did the work and returned the result to the client via a second message. This modular structure made it easier to port it to several computers besides the Intel line, including the DEC Alpha, IBM PowerPC, and SGI MIPS. However, for performance reasons, starting with NT 4.0, pretty much all of

the system was put back into the kernel.

One could go on for a long time both about how NT is structured internally and what its system call interface is like. Since our primary interest here is the virtual machine presented by various operating systems (i.e., the system calls), we will give a brief summary of the system structure and then move on to the system call interface.

The structure of NT is illustrated in Fig. 6-3. It consists of a number of modules that are structured in layers and work together to implement the operating system. Each module has some particular function and a well-defined interface to the other modules. Nearly all the modules are written in C, although part of the graphics device interface is written in C++ and a tiny bit of the lowest layers are written in assembly language.

At the bottom is a thin layer called the **hardware abstraction layer**. Its job is to present the rest of the operating system with abstract hardware devices, devoid of the warts and idiosyncracies with which real hardware is so richly endowed. Among the devices modeled are off-chip caches, timers, I/O buses, interrupt controllers, and DMA controllers. By exposing these to the rest of the operating system in idealized form, it becomes easier to port NT to other hardware platforms, since most of the modifications required are concentrated in one place.

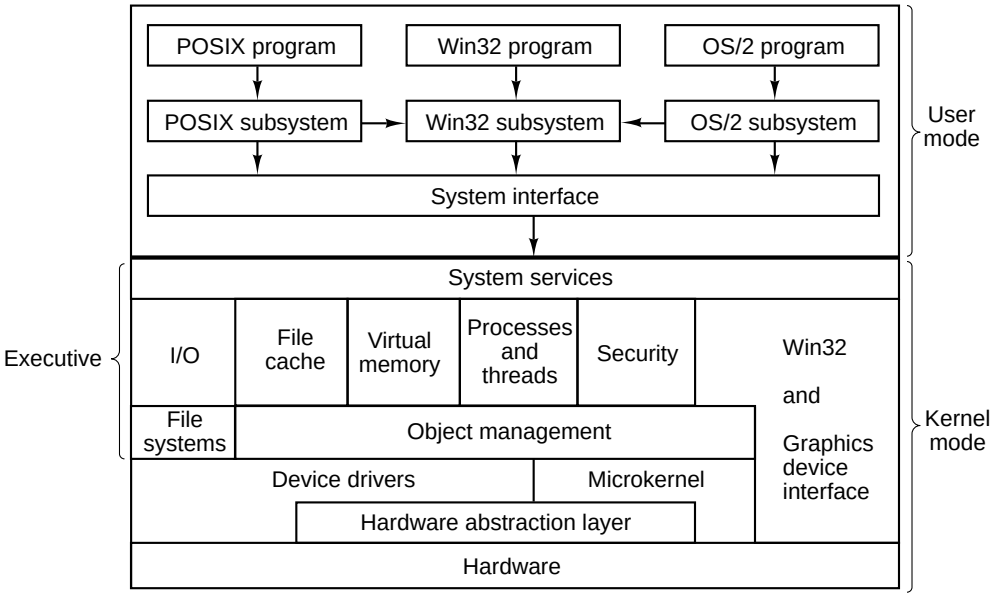


Figure 6-3. The structure of Windows NT.

Above the hardware abstraction layer is a layer containing the microkernel. and the device drivers. The microkernel and all the device drivers have direct access to the hardware when needed, as they contain hardware-dependent code.

The **microkernel** supports the primitive kernel objects, interrupt, trap, and

exception handling, process scheduling and synchronization, multiprocessor synchronization, and time management. The purpose of this layer is to make the rest of the operating system completely independent of the hardware, and thus highly portable. The microkernel is permanently resident in main memory and is not preemptable, although it can temporarily give up control to service I/O interrupts.

Each **device driver** can control one or more I/O devices, but a device driver can also do things not related to a specific device, such as encrypting a data stream or even just providing access to kernel data structures. Since users can install new device drivers, they have the power to affect the kernel and corrupt the system. For this reason, drivers must be written with great care.

Above the microkernel and device drivers is the upper portion of the operating system, called the **executive**. The executive is architecture independent and can be ported to new machines with relatively little effort. It consists of three layers.

The lowest layer contains the file systems and the object manager. The **file systems** support the use of files and directories The **object manager** handles objects known to the kernel. These include processes, threads (lightweight processes within an address space), files, directories, semaphores, I/O devices, timers, and many others. It also manages a namespace in which newly created objects can be placed so they can be referred to later.

The next layer consists of six major parts, as shown in Fig. 6-3. The **I/O manager** provides a framework for managing I/O devices and provides generic I/O services. It uses the services of the file system, which in turn uses the device drivers, as well as the services of the object manager.

The **cache manager** keeps the most recently used disk blocks in memory to speed up access to them in the (likely) event that they are needed again. Its job is to figure out which blocks are probably going to be needed again and which ones are not. It is possible to configure NT with multiple file systems, in which case the cache manager works for all of them, so each one does not have to do its own cache management. When a block is needed, the cache manager is asked to supply it. If it does not have the block, the cache manager calls upon the appropriate file system to get it. Since files can be mapped into processes' address spaces, the cache manager must interact with the virtual memory manager to provide the necessary consistency.

The **virtual memory manager** implements NT's demand-paged virtual memory architecture. It manages the mapping of virtual pages onto physical page frames. It thereby enforces the protection rules that restrict each process to only access those pages belonging to its address space and not to other processes, address spaces (except under special circumstances). It also handles certain system calls that relate to virtual memory.

The **process and thread manager** handles processes and threads, including their creation and destruction. It is concerned about the mechanisms used to manage them, rather than policies about how they are used.

The **security manager** enforces NT's elaborate security mechanism, which

meets the U.S. Dept. of Defense's Orange Book C2 requirements. The Orange Book specifies a large number of rules that a conforming system must meet, starting with authenticated login through how access control is handled to the fact that virtual pages must be zeroed out before being reused.

The **graphics device interface** handles image management for the monitor and printers. It provides system calls to allow user programs to write on the monitor or printers in a device-independent way. It also contains the window manager and hardware device drivers. In versions of NT prior to NT 4.0, it was in user space but the performance was disappointing, so Microsoft moved it into the kernel to speed it up. The Win32 module also handles most of the system calls. It, too, was originally in user space but were also moved to the kernel to improve performance.

On the top of the executive is a thin layer called **system services**. Its function is to provide an interface to the executive. It accepts the true NT system calls and calls other parts of the executive to have them executed.

Outside the kernel are the user programs and the environmental subsystems. The **environmental subsystems** are provided because user programs are not encouraged to make system calls directly (although they are technically capable of it). Instead, each environmental subsystem exports a (different) set of function calls that user programs can use. Three environmental subsystems are shown in Fig. 6-3: Win32 (for NT or Windows 95/98 programs), POSIX (for UNIX programs that have been ported), and OS/2 (for OS/2 programs that have been ported).

Windows applications use the Win32 functions and communicate with the Win32 subsystem to make system calls. The **Win32 subsystem** accepts the Win32 function calls and uses the **system interface** library module (actually, a DLL file—see Chap. 7) to make the necessary true NT system calls to carry them out.

There is also a **POSIX subsystem**, which provides minimal support for UNIX applications. It supports only the P1003.1 functionality and little else. It is a closed subsystem, meaning that its applications cannot use the facilities of the Win32 subsystem, which greatly restricts what they can do. In practice, porting any real UNIX program to NT using this subsystem is nearly impossible. It was included only because parts of the U.S. government require operating systems for government computers to be P1003.1 compliant. This subsystem is not self-contained so it uses the Win32 subsystem for some of its work, but without exporting the full Win32 interface to its user programs.

The OS/2 subsystem is similarly limited in functionality and may be dropped in some future release. It also uses the Win32 subsystem. There is also an MS-DOS environmental subsystem (not shown in the figure).

Having looked briefly at the structure of NT, let us now turn to our main subject, the services offered by NT. This interface is the programmer's main connection to the system. Unfortunately, Microsoft has never made the complete list of NT system calls public, and it also changes them from release to release. Under

such conditions, writing programs that make system calls directly is nearly impossible.

What Microsoft has done is define a set of calls called the **Win32 API** (**Application Programming Interface**) that are publicly known. These are library procedures that either make system calls to get the work done, or, in some case, do the work right in the user-space library procedure or in the Win32 subsystem. The Win32 API calls do not change with new releases, to promote stability. However, there are also NT API calls that may change between releases of NT. Although the Win32 API calls are not all NT system calls, it is better to focus on these here rather than the true NT system calls because the Win32 API calls are well documented are more stable over time.

The Win32 API philosophy is completely different from the UNIX philosophy. In the latter, the system calls are all publicly known and form a minimal interface: removing even one of them would reduce the functionality of the operating system. The Win32 philosophy is to provide a very comprehensive interface, often with three or four ways of doing the same thing, and including many functions that clearly should not be (and are not) system calls, such as an API call to copy an entire file.

Many Win32 API calls create kernel objects of one kind or another, including files, processes, threads, pipes, etc. Every call creating a kernel object returns a result called a **handle** to the caller. This handle can be subsequently used to perform operations on the object. Handles are specific to the process that created the object referred to by the handle. They cannot be passed directly to another process and used there (just as UNIX file descriptors cannot be passed to other processes and used there). However, under certain circumstances, it is possible to duplicate a handle and pass it to other processes in a protected way, allowing them controlled access to objects belonging to other processes. Every object also has a security descriptor associated with it, telling in detail who may and may not perform what kinds of operations on the object.

NT is sometimes said to be object-oriented because the only way to manipulate kernel objects is by invoking methods (API functions) on their handles. On the other hand, it lacks some of the most basic properties of object-oriented systems such as inheritance and polymorphism.

The Win32 API is also available on Windows 95/98 (as well as on the consumer electronics operating system, Windows CE), with a small number of exceptions. For example, Windows 95/98 do not have any security, so those API calls that relate to security just return error codes on Windows 95/98. Also, NT file names use the Unicode character set, which is not available on Windows 95/98. There are also differences in parameters to some API function calls. On NT, for example, all the screen coordinates given in the graphics functions are true 32-bit numbers; on Windows 95/98, only the low-order 16 bits are used (for backward compatibility with Windows 3.1). The existence of the Win32 API on several different operating systems makes it easier to port programs between them but also

points out more clearly that it is somewhat decoupled from the actual system calls. Some of the differences between Windows 95/98 and NT are summarized in Fig. 6-4.

Item	Windows 95/98	NT 5.0
Win32 API?	Yes	Yes
Full 32-bit system?	No	Yes
Security?	No	Yes
Protected file mappings?	No	Yes
Private address space for each MS-DOS program?	No	Yes
Plug and play?	Yes	Yes
Unicode?	No	Yes
Runs on	Intel 80x86	80x86, Alpha
Multiprocessor support?	No	Yes
Re-entrant code inside OS?	No	Yes
Some critical OS data writable by user?	Yes	No

Figure 6-4. Some differences between versions of Windows.

6.4.2 Examples of Virtual Memory

In this section we will look at virtual memory in both UNIX and NT. For the most part, they are fairly similar from the programmer's point of view.

UNIX Virtual Memory

The UNIX memory model is simple. Each process has three segments: code, data, and stack, as illustrated in Fig. 6-5. In a machine with a single, linear address space, the code is generally placed near the bottom of memory, followed by the data. The stack is placed at the top of memory. The code size is fixed, but the data and stack may each grow, in opposite directions. This model is easy to implement on almost any machine and is the model used by Solaris.

Furthermore, if the machine has paging, the entire address space can be paged, without user programs even being aware of it. The only thing they notice is that it is permitted to have programs larger than the machine's physical memory. UNIX systems that do not have paging generally swap entire processes between memory and disk to allow an arbitrarily large number of processes to be timeshared.

For Berkeley UNIX, the above description (demand-paged virtual memory) is basically the entire story. However, System V (and Solaris) include several features that allow users to manage their virtual memory in more sophisticated

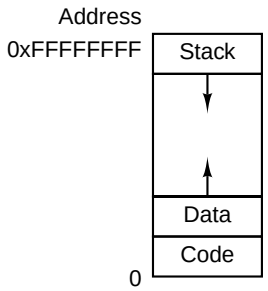


Figure 6-5. The address space of a single UNIX process.

ways. Most important of these is the ability of a process to map a (portion of) a file onto part of its address space. For example, if a 12-KB file is mapped at virtual address 144K, a read to the word at address 144K reads the first word of the file. In this way file I/O can be done without making system calls. Since some files may exceed the size of the virtual address space, it is also possible to map in only a portion of a file instead of the whole file. The mapping is done first opening the file and getting back a file descriptor, *fd*, which is used to identify the file to be mapped. Then the process makes a call

```
paddr = mmap(virtual_address, length, protection, flags, fd, file_offset)
```

which maps *length* bytes starting at *file_offset* in the file onto the virtual address space starting at *virtual_address*. Alternatively, the *flags* parameter can be set to ask the system to choose a virtual address, which it then returns as *paddr*. The mapped region must be an integral number of pages and aligned at a page boundary. The *protection* parameter can specify any combination of read, write, or execute permission. The mapping can be removed later with unmap.

Multiple processes can map onto the same file at the same time. Two options are provided for sharing. In the first one, all the pages are shared, so writes done by one process are visible to all the others. This option provides a high-bandwidth communication path between processes. The other option shares the pages as long as no process modifies them. However, as soon as any process attempts to write on a page, it gets a protection fault, which causes the operating system to give it a private copy of the page to write on. This scheme, known as **copy on write**, is used when each of multiple processes needs the illusion it is the only one mapped onto a file.

Windows NT Virtual Memory

In NT, every user process has its own virtual address space. Virtual addresses are 32 bits long, so each process has 4 GB of virtual address space. The lower 2 GB are available for the process' code and data; the upper 2 GB allow (limited) access to kernel memory, except in Enterprise versions of NT in which the split

can be 3 GB for the user and 1 GB for the kernel. The virtual address space is demand paged, with a fixed page size (4 KB on the Pentium II).

Each virtual page can be in one of three states: free, reserved, or committed. A **free page** is not currently [Hin use and a reference to it causes a page fault. When a process is started, all of its pages are in free state until the program and initial data are mapped into its address space. Once code or data is mapped onto a page, the page is said to be **committed**. A reference to a committed page is mapped using the virtual memory hardware and succeeds if the page is in main memory. If the page is not in main memory, a page fault occurs and the operating system finds and brings in the page from disk. A virtual page can also be in **reserved** state, meaning it is not available for being mapped until the reservation is explicitly removed. In addition to the free, reserved, and committed attributes, pages also have other attributes, such as being readable, writable, and executable. The top 64 KB and bottom 64 KB of memory are always free, to catch pointer errors (uninitialized pointers are often 0 or –1).

Each committed page has a shadow page on the disk where it is kept when it is not in main memory. Free and reserved pages do not have shadow pages, so references to them cause page faults (the system cannot bring in a page from disk if there is no page on disk). The shadow pages on the disk are arranged into one or more paging files. The operating system keeps track of which virtual page maps onto which part of which paging file. For (execute only) program text, the executable binary file contains the shadow pages; for data pages, special paging files are used.

NT, like System V, allows files to be mapped directly onto regions of the virtual address spaces (i.e., runs of consecutive pages). Once a file has been mapped onto the address space, it can be read or written using ordinary memory references.

Memory-mapped files are implemented in the same way as other committed pages, only the shadow pages can be in the disk file instead of in the paging file. As a result, when a file is mapped in, the version in memory may not be identical to the disk version (due to recent writes to the virtual address space). However, when the file is unmapped or is explicitly flushed, the disk version is brought up to date.

NT explicitly allows two or more processes to map in the same file at the same time, possibly at different virtual addresses. By reading and writing memory words, the processes can now communicate with each other and pass data back and forth at very high bandwidth, since no copying is required. Different processes may have different access permissions. Since all the processes using a mapped file share the same pages, changes made by one of them are immediately visible to all the others, even if the disk file has not yet been updated.

The Win32 API contains a number of functions that allow a process to manage its virtual memory explicitly. The most important of these functions are listed in Fig. 6-6. All of them operate on a region consisting either of a single

page or a sequence of two or more pages that are consecutive in the virtual address space.

API function	Meaning
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it)
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space
UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file mapping object

Figure 6-6. The principal API functions for managing virtual memory in Windows NT.

The first four API functions are self-explanatory. The next two give a process the ability to hardwire up to 30 pages in memory so they will not be paged out and to undo this property. A real-time program might need this ability, for example. A limit is enforced by the operating system to prevent processes from getting too greedy. Although not shown in Fig. 6-6, NT also has API functions to allow a process to access the virtual memory of a different process over which it has been given control (i.e., for which it has a handle).

The last four API functions listed are for managing memory-mapped files. To map a file, a file mapping object must first be created, with CreateFileMapping. This function returns a handle to the file mapping object and optionally enters a name for it into the file system so another process can use it. The next two functions map and unmap files, respectively. The last one can be used by a process to map in a file currently also mapped in by a different process. In this way, two or more processes can share regions of their address spaces.

These API functions are the basic ones upon which the rest of the memory management system is built. For example, there are API functions for allocating and freeing data structures on one or more heaps. Heaps are used for storing data structures that are dynamically created and destroyed. The heaps are not garbage collected, so it is up to user software to free blocks of virtual memory that are no longer in use. (Garbage collection is the automatic removal of unused data structures by the system.) Heap usage in NT is similar to the use of the *malloc* function in UNIX systems, except that there can be multiple independently managed heaps.

6.4.3 Examples of Virtual I/O

The heart of any operating system is providing services to user programs, mostly I/O services such as reading and writing files. Both UNIX and NT offer a wide variety of I/O services to user programs. For most UNIX system calls, NT has an equivalent call, but the reverse is not true as NT has far more calls and each of them is far more complicated than its UNIX counterpart.

UNIX Virtual I/O

Much of the popularity of the UNIX system can be traced directly to its simplicity, which, in turn, is a direct result of the organization of the file system. An ordinary file is a linear sequence of 8-bit bytes starting at 0 and going up to a maximum of $2^{32} - 1$ bytes. The operating system itself imposes no record structure on files, although many user programs regard ASCII text files as sequences of lines, each line terminated by a line feed.

Associated with every open file is a pointer to the next byte to be read or written. The read and write system calls read and write data starting at the file position indicated by the pointer. Both calls advance the pointer after the operation by an amount equal to the number of bytes transferred. However, random access to files is possible by explicitly setting the file pointer to a specific value.

In addition to ordinary files, the UNIX system also supports special files, which are used to access I/O devices. Each I/O device typically has one or more special files assigned to it. By reading and writing from the associated special file, a program can read or write from the I/O device. Disks, printers, terminals, and many other devices are handled this way.

The major UNIX file system calls are listed in Fig. 6-7. The creat call (without the *e*) can be used to create a new file. It is not strictly necessary any more, because open can also create a new file now. Unlink removes a file, assuming that the file is in only one directory.

Open is used to open existing files (and create new ones). The mode flag tells how to open it (for reading, for writing, etc.). The call returns a small integer called a **file descriptor** that identifies the file in subsequent calls. The actual file I/O is done with read and write, each of which has a file descriptor indicating which file to use, a buffer for the data to go to or come from, and a byte count telling how much data to transmit. Lseek is used to position the file pointer, making random access to files possible.

Stat returns information about a file, including its size, time of last access, owner, and more. Chmod changes the protection mode of a file, for example, allowing or forbidding users other than the owner from reading it. Finally, fcntl does various miscellaneous operations on a file, such as locking or unlocking it.

Figure 6-8 illustrates how the major file I/O calls work. This code is minimal and does not include the necessary error checking. Before entering the loop, the

System call	Meaning
creat(name, mode)	Create a file; *mode* specifies the protection mode
unlink(name)	Delete a file (assuming that there is only 1 link to it)
open(name, mode)	Open or create a file and return a file descriptor
close(fd)	Close a file
read(fd, buffer, count)	Read *count* bytes into *buffer*
write(fd, buffer, count)	Write *count* bytes from *buffer*
lseek(fd, offset, w)	Move the file pointer as required by *offset* and *w*
stat(name, buffer)	Return information about a file
chmod(name, mode)	Change the protection mode of a file
fcntl(fd, cmd, ...)	Do various control operations such as locking (part of) a file

Figure 6-7. The principal UNIX file system calls.

program opens an existing file, *data*, and creates a new file, *newf*. Each call returns a file descriptor, *infd*, and *outfd*, respectively. The second parameters to the two calls are protection bits that specify that the files are to be read and written, respectively. Both calls return a file descriptor. If either open or creat fails, a negative file descriptor is returned, telling that the call failed.

```
// Open the file descriptors
infd = open(″data″, 0);
outfd = creat(″newf″, ProtectionBits);

// Copy loop
do {
        count = read(infd, buffer, bytes);
        if (count > 0) write(outfd, buffer, count);
} while (count > 0);

// Close the files
close(infd);
close(outfd);
```

Figure 6-8. A program fragment for copying a file using the UNIX system calls. This fragment is in C because Java hides the low-level system calls and we are trying to expose them.

The call to read has three parameters: a file descriptor, a buffer, and a byte count. The call tries to read the desired number of bytes from the indicated file into the buffer. The number of bytes actually read is returned in *count*, which will be smaller than *bytes* if the file was too short. The write call deposits the newly read bytes on the output file. The loop continues until the input file has been completely read, at which time the loop terminates and both files are closed.

File descriptors in UNIX are small integers (usually below 20). File descriptors 0, 1, and 2 are special and correspond to **standard input**, **standard output**, and **standard error**, respectively. Normally, these refer to the keyboard, the display, and the display, respectively, but they can be redirected to files by the user. Many UNIX programs get their input from standard input and write the processed output on standard output. Such programs are often called **filters**.

Closely related to the file system is the directory system. Each user may have multiple directories, with each directory containing both files and subdirectories. UNIX systems normally are configured with a main directory, called the **root directory**, containing subdirectories *bin* (for frequently executed programs), *dev* (for the special I/O device files), *lib* (for libraries), and *usr* (for user directories), as shown in Fig. 6-9. In this example, the *usr* directory contains subdirectories for *ast* and *jim*. The *ast* directory contains two files, *data* and *foo.p*, and a subdirectory, *bin*, containing four games.

Files can be named by giving their **path** from the root directory. A path contains a list of all the directories traversed from the root to the file, with directory names separated by slashes. For example, the absolute path name of *game2* is */usr/ast/bin/game2*. A path starting at the root is called an **absolute path**.

At every instant, each running program has a **working directory**. Path names may also be relative to the working directory, in which case they do not begin with a slash, to distinguish them from absolute path names. Such paths are called **relative paths**. When */usr/ast* is the working directory, *game3* can be accessed using the path *bin/game3*. A user may create a **link** to someone else's file using the link system call. In the above example, */usr/ast/bin/game3* and */usr/jim/jotto* both access the same file. To prevent cycles in the directory system, links are not permitted to directories. The calls and creat take either absolute or relative path names as arguments.

The major directory management system calls in UNIX are listed in Fig. 6-10. Mkdir creates a new directory and rmdir deletes an existing (empty) directory. The next three calls are used to read directory entries. The first one opens the directory, the next one reads entries from it, and the last one closes the directory. Chdir changes the working directory.

Link makes a new directory entry with the new entry pointing to an existing file. For example, the entry */usr/jim/jotto* might have been created by the call

```
link("/usr/ast/bin/game3", "/usr/jim/jotto")
```

or an equivalent call using relative path names, depending on the working directory of the program making the call. Unlink removes a directory entry. If the file has only one link, the file is deleted. If it has two or more links, it is kept. It does not matter whether a removed link is the original or a copy made later. Once a link is made, it is a first-class citizen, indistinguishable from the original. The call

```
unlink("/usr/ast/bin/game3")
```

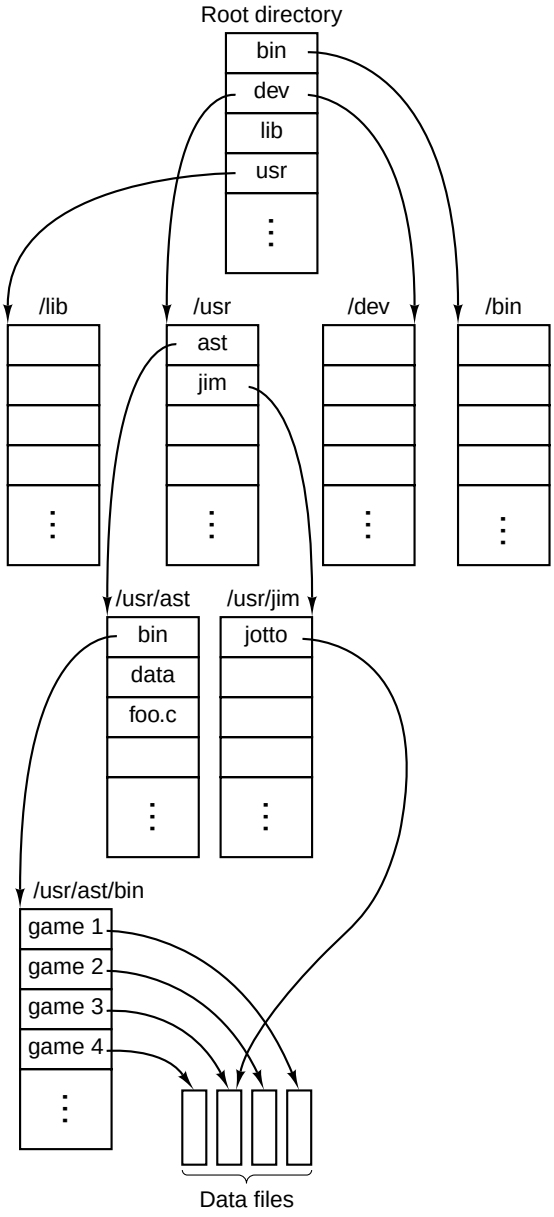


Figure 6-9. Part of a typical UNIX directory system.

makes *game3* only accessible via the path */usr/jim/jotto* henceforth. Link and unlink can be used in this way to ''move'' files from one directory to another.

Associated with every file (including directories, because they are also files) is a bit map telling who may access the file. The map contains three RWX fields,

System call	Meaning
mkdir(name, mode)	Create a new directory
rmdir(name)	Delete an empty directory
opendir(name)	Open a directory for reading
readdir(dirpointer)	Read the next entry in a directory
closedir(dirpointer)	Close a directory
chdir(dirname)	Change working directory to *dirname*
link(name1, name2)	Create a directory entry *name2* pointing to *name1*
unlink(name)	Remove *name* from its directory

Figure 6-10. The principal UNIX directory management calls.

the first controlling the Read, Write, eXecute permissions for the owner, the second for others in the owner's group, and the third for everybody else. Thus RWX R-X --X means that the owner can read the file, write the file, and execute the file (obviously, it is an executable program, or execute would be off), whereas others in his group can read or execute it and strangers can only execute it. With these permissions, strangers can use the program but not steal (copy) it because they do not have read permission. The assignment of users to groups is done by the system administrator, usually called the **superuser**. The superuser also has the power to override the protection mechanism and read, write, or execute any file.

Let us now briefly examine how files and directories are implemented in UNIX. For a more complete treatment, see (Vahalia, 1996). Associated with each file (and each directory, because a directory is also a file) is a 64-byte block of information called an **i-node**. The i-node tells who owns the file, what the permissions are, where to find the data, and similar things. The i-nodes for the files on each disk are located either in numerical sequence at the beginning of the disk, or if the disk is split up into groups of cylinders, at the start of a cylinder group. Thus given an i-node number, the UNIX system can locate the i-node by simply calculating its disk address.

A directory entry consists of two parts: a file name and an i-node number. When a program executes

```
open(″foo.c″, 0)
```

the system searches the working directory for the file name, ''foo.c,'' in order to locate the i-node number for that file. Having found the i-node number, it can then read in the i-node, which tells it all about the file.

When a longer path name is specified, the basic steps outlined above are repeated several times until the full path has been parsed. For example, to locate the i-node number for */usr/ast/data*, the system first searches the root directory for

an entry *usr*. Having found the i-node for *usr*, it can read that file (a directory is a file in UNIX). In this file it looks for an entry *ast*, thus locating the i-node number for the file */usr/ast*. By reading */usr/ast*, the system can then find the entry for *data*, and thus the i-node number for */usr/ast/data*. Given the i-node number for the file, it can then find out everything about the file from the i-node.

The format, contents, and layout of an i-node vary somewhat from system to system (especially when networking is in use), but the following items are typically found in each i-node.

1. The file type, the 9 RWX protection bits, and a few other bits.
2. The number of links to the file (number of directory entries for it).
3. The owner's identity.
4. The owner's group.
5. The file length in bytes.
6. Thirteen disk addresses.
7. The time the file was last read.
8. The time the file was last written.
9. The time the i-node was last changed.

The file type distinguishes ordinary files, directories, and two kinds of special files, for block-structured and unstructured I/O devices, respectively. The number of links and the owner identification have already been discussed. The file length is a 32-bit integer giving the highest byte that has a value. It is perfectly legal to create a file, do an lseek to position 1,000,000, and write 1 byte, which yields a file of length 1,000,001. The file would *not*, however, require storage for all the ''missing'' bytes.

The first 10 disk addresses point to data blocks. With a block size of 1024 bytes, files up to 10,240 bytes can be handled this way. Address 11 points to a disk block, called an **indirect block**, which contains 256 disk addresses. Files up to $10{,}240 + 256 \times 1024 = 272{,}384$ bytes are handled this way. For still larger files, address 12 points to a block containing the addresses of 256 indirect blocks, which takes care of files up to $272{,}384 + 256 \times 256 \times 1024 = 67{,}381{,}248$ bytes. If this **double indirect block** scheme is still too small, disk address 13 is used to point to a **triple indirect block** containing the addresses of 256 double indirect blocks. Using the direct, single, double, and triple indirect addresses, up to 16,843,018 blocks can be addressed giving a theoretical maximum file size of 17,247,250,432 bytes. Since file pointers are limited to 32 bits, the practical upper limit is actually 4,294,967,295 bytes. Free disk blocks are kept on a linked list. When a new block is needed, the next block is plucked from the list. As a result, the blocks of each file are scattered randomly around the disk.

To make disk I/O more efficient, when a file is opened, its i-node is copied to a table in main memory and is kept there for handy reference as long as the file remains open. In addition, a pool of recently referenced disk blocks is maintained in memory. Because most files are read sequentially, it often happens that a file reference requires the same disk block as the previous reference. To strengthen this effect, the system also tries to read the *next* block in a file, before it is referenced, in order to speed up processing. All this optimization is hidden from the user; when a user issues a read call, the program is suspended until the requested data are available in the buffer.

With this background information, we can now see how file I/O works. Open causes the system to search the directories for the specified path. If the search is successful, the i-node is read into an internal table. Reads and writes require the system to compute the block number from the current file position. The disk addresses of the first 10 blocks are always in main memory (in the i-node); higher-numbered blocks require one or more indirect blocks to be read first. Lseek just changes the current position pointer without doing any I/O.

Link and unlink are also simple to understand now. Link looks up its first argument to find the i-node number. Then it creates a directory entry for the second argument, putting the i-node number of the first file in that entry. Finally, it increases the link count in the i-node by one. Unlink removes a directory entry and decrements the link count in the i-node. If it is zero, the file is removed and all the blocks are put back on the free list.

Windows NT Virtual I/O

NT supports several file systems, the most important of which are **NTFS** (**NT File System**) and the **FAT** (**File Allocation Table**) file system. The former is a new file system developed specifically for NT; the latter is the old MS-DOS file system, which is also used on Windows 95/98 (albeit with support for longer file names). Since the FAT file system is basically obsolete (although several hundred million people are still using it), we will study NTFS below. Starting with NT 5.0, the FAT32 file system used in later versions of Windows 95 and in Windows 98 was also supported.

File names in NTFS can be up to 255 characters long. File names are in Unicode, allowing people in countries not using the Latin alphabet (e.g., Japan, India, and Israel) to write file names in their native language. (In fact, NT uses Unicode throughout internally; versions starting with NT 5.0 have a single binary that can be used in any country and still use the local language because all the menus, error messages, etc., are kept in country-dependent configuration files.) NTFS fully supports case-sensitive names (so *foo* is different from *FOO*). Unfortunately, the Win32 API does not fully support case-sensitivity for file names and not at all for directory names, so this advantage is lost to programs using Win32.

As with UNIX, a file is just a linear sequence of bytes, although up to a maximum of $2^{64} - 1$. File pointers also exist, as in UNIX, but are 64 bits wide rather than 32 bits, to handle the maximum length file. The Win32 API function calls for file and directory manipulation are roughly similar to their UNIX counterparts, except most have more parameters and the security model is different. Opening a file returns a handle, which is then used for reading and writing the file. However, unlike in UNIX, handles are not small integers, and standard input, standard output, and standard error have to be acquired explicitly rather than being predefined as 0, 1, and 2 (except in console mode, where they are preopened). The principal Win32 API functions for file management are listed in Fig. 6-11.

API function	UNIX	Meaning
CreateFile	open	Create a file or open an existing file; return a handle
DeleteFile	unlink	Destroy an existing file
CloseHandle	close	Close a file
ReadFile	read	Read data from a file
WriteFile	write	Write data to a file
SetFilePointer	lseek	Set the file pointer to a specific place in the file
GetFileAttributes	stat	Return the file properties
LockFile	fcntl	Lock a region of the file to provide mutual exclusion
UnlockFile	fcntl	Unlock a previously locked region of the file

Figure 6-11. The principal Win32 API functions for file I/O. The second column gives the nearest UNIX equivalent.

Let us now examine these calls briefly. CreateFile can be used to create a new file and return a handle to it. This API function is also used to open existing files as there is no open API function. We have not listed the parameters for the NT API functions because they are so voluminous. As an example, CreateFile has seven parameters, as follows:

1. A pointer to the name of the file to create or open.
2. Flags telling whether the file can be read, written, or both.
3. Flags telling whether multiple processes can open the file at once.
4. A pointer to the security descriptor, telling who can access the file.
5. Flags telling what to do if the file exists/does not exist.
6. Flags dealing with attributes such as archiving, compression, etc.
7. The handle of a file whose attributes should be cloned for the new file.

The next six API functions in Fig. 6-11 are fairly similar to the corresponding UNIX system calls. The last two allow a region of a file to be locked and unlocked to permit a process to get guaranteed mutual exclusion to it.

Using these API functions, it is possible to write a procedure to copy a file, analogous to the UNIX version of Figure 6-1. Such a procedure (without any error checking) is shown in Fig. 6-12. It has been designed to mimic the structure of Figure 6-8. In practice, one would not have to program a copy file program since CopyFile is an API function (which executes something close to this program as a library procedure).

```
// Open files for input and output.
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);

// Copy the file.
do {
        s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
        if (s > 0 && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
while (s > 0 && count > 0);

// Close the files.
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Figure 6-12. A program fragment for copying a file using the Windows NT API functions. This fragment is in C because Java hides the low-level system calls and we are trying to expose them.

NT supports a hierarchical file system, similar to the UNIX file system. The separator between component names is \ however, instead of /, a fossil inherited from MS-DOS. There is a concept of a current working directory and path names can be relative or absolute. One significant difference with UNIX, however, is UNIX allows the file systems on different disks and machines to be mounted together in a single naming tree, thus hiding the disk structure from all software. NT 4.0 does not have this property, so absolute file names must begin with a drive letter indicating which logical disk is meant, as in *C:\windows\system\foo.dll*. Starting with NT 5.0, UNIX-style 5.0 mounting of file systems was added.

The major directory management API functions are given in Fig. 6-13, again along with their nearest UNIX equivalents. The functions are hopefully self-explanatory.

Note that NT 4.0 did not support file links. At the level of the graphical desktop, shortcuts were supported, but these constructions were internal to the desktop and had no counterpart in the file system itself. An application program could not enter a file in a second directory without copying the entire file. Starting with NT 5.0 file linking was added to the file system proper.

API function	UNIX	Meaning
CreateDirectory	mkdir	Create a new directory
RemoveDirectory	rmdir	Remove an empty directory
FindFirstFile	opendir	Initialize to start reading the entries in a directory
FindNextFile	readdir	Read the next directory entry
MoveFile		Move a file from one directory to another
SetCurrentDirectory	chdir	Change the current working directory

Figure 6-13. The principal Win32 API functions for directory management. The second column gives the nearest UNIX equivalent, when one exists.

NT has a much more elaborate security mechanism than UNIX. Although there are hundreds of API functions relating to security, the following brief description gives the general idea. When a user logs in, his or her initial process is given an **access token** by the operating system. The access token contains the user's **SID** (**Security ID**), a list of the security groups to which the user belongs, any special privileges available, and a few other items. The point of the access token is to concentrate all the security information in one easy-to-find place. All processes created by this process inherit the same access token.

One of the parameters that can be supplied when any object is created is its **security descriptor**. The security descriptor contains a list of entries called an **ACL** (**Access Control List**). Each entry permits or prohibits some set of the operations on the object by some SID or group. For example, a file could have a security descriptor specifying that Elinor has no access to the file at all, Ken can read the file, Linda can read or write the file, and that all members of the XYZ group can read the file's length but nothing else.

When a process tries to perform some operation on an object using the handle it got when it opened the object, the security manager gets the process' access token and goes down the list of entries in the ACL in order. As soon as it finds an entry that matches the caller's SID or one of the caller's groups, the access found there is taken as definitive. For this reason, it is usual to put entries denying access ahead of entries granting access in the ACL, so that a user who is specifically denied access cannot get in via a back door by being a member of a group that has legitimate access. The security descriptor also contains information used for auditing accesses to the object.

Let us now take a quick look at how file and directories are implemented in NT. Each disk is statically divided up into self-contained volumes, which are the same as disk partitions in UNIX. Each volume contains files, directories bit maps, and other data structures for managing its information. Each volume is organized as a linear sequence of **clusters**, with the cluster size being fixed for each volume and ranging from 512 bytes to 64 KB, depending on the volume size. Clusters are referred to by their offset from the start of the volume using 64-bit numbers.

The main data structure in each volume is the **MFT** (**Master File Table**), which has an entry for each file and directory in the volume. These entries are analogous to the i-nodes in UNIX. The MFT is itself a file, and as such can be placed anywhere within the volume, thus eliminating the problem that UNIX has with bad disk blocks in the middle of the i-nodes.

The MFT is shown in Fig. 6-14. It begins with a header containing information about the volume, such as (pointers to) the root directory, the boot file, the bad-block file, the free-list administration, etc. After that comes an entry per file or directory, 1 KB except when the cluster size is 2 KB or more. Each entry contains all the metadata (administrative information) about the file or directory. Several formats are allowed, one of which is shown in Fig. 6-14.

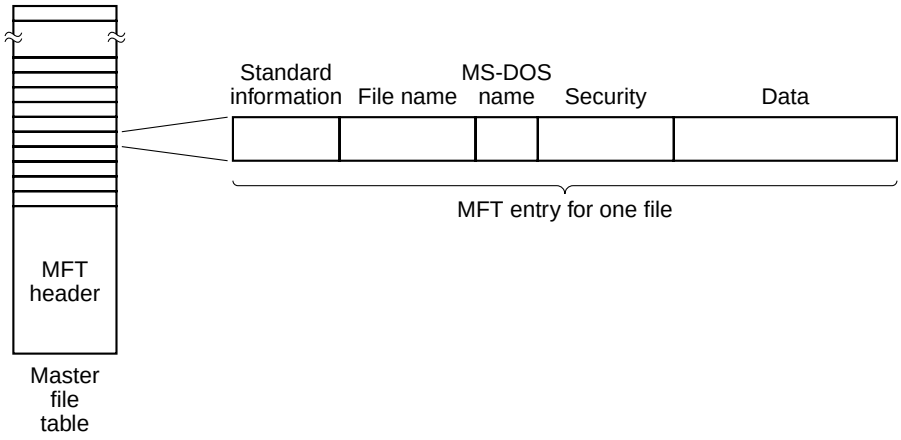


Figure 6-14. The Windows NT master file table.

The standard information field contains information such as the time stamps needed by POSIX, the hard link count, the read-only and archive bits, etc. It is a fixed-length field and always present. The file name is variable length, up to 255 Unicode characters. In order to make such files accessible to old 16-bit programs, files can also have a MS-DOS name, which consists of 8 alphanumeric characters optionally followed by a dot and an extension of up to 3 alphanumeric characters. If the actual file name conforms to the MS-DOS 8+3 naming rule, a secondary MS-DOS name is not used.

Next comes the security information. In versions up to and including NT 4.0, the security field contained the actual security descriptor. Starting with NT 5.0, all the security information was centralized in a single file, with the security field simply pointing to the relevant part of this file.

For small files, the file data itself is actually contained in the MFT entry, saving a disk access to fetch it. This idea is called an **immediate file** (Mullender and Tanenbaum, 1987). For somewhat larger files, this field contains pointers to the clusters containing the data, or more commonly, runs of consecutive clusters so a

single cluster number and a length can represent an arbitrary amount of file data. If a single MFT entry is insufficiently large to hold whatever information it is supposed to hold, one or more additional entries can be chained to it.

The maximum file size is 2^{64} bytes. To get an idea of how big a 2^{64}-byte file is, imagine that the file were written out in binary, with each 0 or 1 occupying 1 mm of space. The 2^{67}-mm listing would be 15 light-years long, reaching far beyond the solar system, to Alpha Centauri and back.

The NTFS file system has many other interesting properties including data compression and fault tolerance using atomic transactions. Additional information about it can be found in (Solomon, 1998).

6.4.4 Examples of Process Management

Both UNIX and NT allow a job to be split up into multiple processes that can run in (pseudo)parallel and communicate with each other, in the style of the producer-consumer example discussed earlier. In this section we will discuss how processes are managed in both systems. Both systems also support parallelism within a single process using threads, so that will also be discussed.

UNIX Process Management

At any time, a UNIX process can create a subprocess that is an exact replica of itself by executing the fork system call. The original process is called the **parent** and the new one is called the **child**. Right after the fork, the two processes are identical and even share the same file descriptors. Thereafter, each one goes its own way and does whatever it wants to, independent of the other one.

In many cases, the child process juggles the file descriptors in certain ways and then executes the exec system call, which replaces its program and data with the program and data found in an executable file specified as parameter to the exec call. For example, when a user types a command *xyz* at a terminal, the command interpreter (shell) executes fork to create a child process. This child process then executes exec to run the *xyz* program.

The two processes run in parallel (with or without exec), unless the parent wishes to wait for the child to terminate before continuing. If the parent wishes to wait, it executes either the wait or waitpid system call, which causes it to be suspended until the child finishes by executing exit. After the child finishes, the parent continues.

Processes can execute fork as often as they want, giving rise to a tree of processes. In Fig. 6-15, for example, process *A* has executed fork twice, creating two children, *B* and *C*. Then *B* also executed fork twice, and *C* executed it once, giving the final tree of six processes.

Processes in UNIX can communicate with each other via a structure called a **pipe**. A pipe is a kind of buffer into which one process can write a stream of data

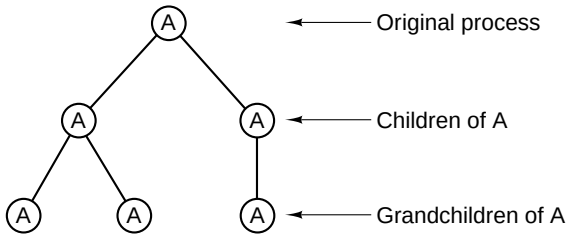


Figure 6-15. A process tree in UNIX.

and another can take it out. Bytes are always retrieved from a pipe in the order they were written. Random access is not possible. Pipes do not preserve message boundaries, so if one process does four 128-byte writes and the other does a 512-byte read, the reader will get all the data at once, with no indication that they were written in multiple operations.

In System V and Solaris, another way for processes to communicate is by using **message queues**. A process can create a new message queue or open an existing one using msgget. Using a message queue, a process can send messages using msgsnd and receive them using msgrecv. Messages sent this way differ from data stuffed into a pipe in several ways. First, message boundaries are preserved, whereas a pipe is just a byte stream. Second, messages have priorities, so urgent ones can skip ahead of less important ones. Third, messages are typed, and a msgrecv can specify a particular type, if desired.

Another communication mechanism is the ability of two or more processes to share a region of their respective address spaces. UNIX handles this shared memory by mapping the same pages into the virtual address space of all the sharing processes. As a result, a write by one process into the shared region is immediately visible to the other processes. This mechanism provides a very high bandwidth communication path between processes. The system calls involved in shared memory go by names like shmat and shmop.

Another feature of System V and Solaris is the availability of semaphores. These work essentially as described in the producer-consumer example given in the text.

Another facility provided by all POSIX-conformant UNIX systems is the ability to have multiple threads of control within a single process. These threads of control, usually just called **threads**, are like lightweight processes that share a common address space and everything associated with that address space, such as file descriptors, environment variables, and outstanding timers. However, each thread has its own program counter, own registers, and own stack. When a thread blocks (i.e., has to stop temporarily until I/O completes or some other event happens), other threads in the same process are still able to run. Two threads in the same process operating as a producer and consumer are similar, but not identical, to two single-thread processes that are sharing a memory segment containing a buffer.

The differences have to do with the fact that in the latter case, each process has its own file descriptors, etc., whereas in the former case all of these items are shared. We saw the use of Java threads in our producer-consumer example earlier. Often the Java runtime system uses a operating system thread for each of its threads, but it does not have to do this.

As an example of where threads might be useful, consider a World Wide Web server. Such a server might keep a cache of commonly-used Web pages in main memory. If a request is for a page in the cache, the Web page is returned immediately. Otherwise, it is fetched from disk. Unfortunately, waiting for the disk takes a long time (typically 20 msec), during which the process is blocked and cannot serve new incoming requests, even those for Web pages in the cache.

The solution is to have multiple threads within the server process, all of which share the common Web page cache. When one thread blocks, other threads can handle new requests. To prevent blocking without threads, one could have multiple server processes, but this would probably entail replicating the cache, thus wasting valuable memory.

The UNIX standard for threads is called **pthreads**, and is defined by POSIX (P1003.1C) It contains calls for managing and synchronizing threads. It is not defined whether threads are managed by the kernel or entirely in user space. The most commonly-used thread calls are listed in Fig. 6-16.

Thread call	Meaning
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Figure 6-16. The principal POSIX thread calls.

Let us briefly examine the thread calls shown in Fig. 6-16. The first call, pthread_create, creates a new thread. After successful completion, one more thread is running in the caller's address space than was before the call. A thread that has done its job and wants to terminate calls pthread_exit. A thread can wait for another thread to exit by calling pthread_join. If the thread waited for has already exited, the pthread_join finishes immediately. Otherwise it blocks.

Threads can synchronize using locks called **mutexes**. Typically a mutex guards some resource, such as a buffer shared by two threads. To make sure that only one thread at a time accesses the shared resource, threads are expected to lock the mutex before touching the resource and unlock it when they are done. As long as all threads obey this protocol, race conditions can be avoided. Mutexes are like binary semaphores, that is, semaphores that can take on only the values of 0 and 1. The name ''mutex'' comes from the fact that mutexes are used to ensure mutual exclusion on some resource.

Mutexes can be created and destroyed by the calls pthread_mutex_init and pthread_mutex_destroy, respectively. A mutex can be in one of two states: locked or unlocked. When a thread tries to lock an unlocked mutex (using pthread_lock), the lock is set and the thread continues. However, when a thread tries to lock a mutex that is already locked, it blocks. When the locking thread is finished with the shared resource, it is expected to unlock the corresponding mutex by calling pthread_mutex_unlock.

Mutexes are intended for short-term locking, such as protecting a shared variable. They are not intended for long-term synchronization, such as waiting for a tape drive to become free. For long-term synchronization, **condition variables** are provided. These are created and destroyed by calls to pthread_cond_init and pthread_cond_destroy, respectively.

A condition variable is used by having one thread wait on it, and another signal it. For example, having discovered that the tape drive it needs is busy, a thread would do pthread_cond_wait on a condition variable that all the threads have agreed to associate with the tape drive. When the thread using the tape drive is finally done with it (possibly hours later), it uses pthread_cond_signal to release exactly one thread waiting on that condition variable (if any). If no thread is waiting, the signal is lost. Condition variables do not count like semaphores. A few other operations are also defined on threads, mutexes, and condition variables.

Windows NT Process Management

NT supports multiple processes, which can communicate and synchronize. Each process contains at least one thread, which in turn contains at least one fiber (lightweight thread). Together, processes, threads, and fibers provide a very general set of tools for managing parallelism, both on uniprocessors (single-CPU machines) and on multiprocessors (multi-CPU machines).

New processes are created using the API function CreateProcess. This function has 10 parameters, each of which has many options. This design is clearly a lot more complicated that the UNIX scheme, in which fork has no parameters, and exec has just three: pointers to the name of the file to execute, the (parsed) command line parameter array, and the environment strings. Roughly speaking, the 10 parameters to CreateProcess are as follows:

1. A pointer to the name of the executable file.
2. The command line itself (unparsed).
3. A pointer to a security descriptor for the process.
4. A pointer to a security descriptor for the initial thread.
5. A bit telling whether the new process inherits the creator's handles.
6. Miscellaneous flags (e.g., error mode, priority, debugging, consoles).
7. A pointer to the environment strings.
8. A pointer to the name of the new process' current working directory.
9. A pointer to a structure describing the initial window on the screen.
10. A pointer to a structure that returns 18 values to the caller.

NT does not enforce any kind of parent-child or other hierarchy. All processes are created equal. However, since 1 of the 18 parameters returned to the creating process is a handle to the new process (allowing considerable control over the new process), there is an implicit hierarchy in terms of who has a handle to whom. Although these handles cannot just be passed directly to other processes, there is a way for a process to make a handle suitable for another process and then give it the handle, so the implicit process hierarchy may not last long.

Each process in NT is created with a single thread, but a process can create more threads later on. Thread creation is simpler than process creation: CreateThread has only six parameters instead of 10: the security descriptor, the stack size, the starting address, a user-defined parameter, the initial state of the thread (ready or blocked), and the thread's ID. The kernel does the thread creation, so it is clearly aware of threads (i.e, they are not implemented purely in user space as is the case in some other systems).

When the kernel does scheduling, it not only picks the process to run next but also which thread in that process. This means that the kernel is always aware of which threads are ready and which ones are blocked. Because threads are kernel objects, they have security descriptors and handles. Since a handle for a thread can be passed to another process, it is possible to have one process control the threads in a different process. This feature is useful for debuggers, for example.

Threads in NT are relatively expensive because doing a thread switch requires entering and later leaving the kernel. To provide very lightweight pseudoparallelism, NT provides **fibers**, which are like threads, but are scheduled in user space by the program that created them (or its runtime system). Each thread can have multiple fibers, the same way a process can have multiple threads, except that when a fiber logically blocks, it puts itself on the queue of blocked fibers and selects another fiber to run in the context of its thread. The kernel is not aware of this

transition because the thread keeps running, even though it may be first running one fiber, then another. The kernel just manages processes and threads, not fibers. Fibers are useful, for example, when programs that manage their own threads are ported to NT.

Processes can communicate in a wide variety of ways, including pipes, named pipes, mailslots, sockets, remote procedure calls, and shared files. Pipes have two modes: byte and message, selected at creation time. Byte-mode pipes work the same way as in UNIX. Message-mode pipes are somewhat similar but preserve message boundaries, so that four writes of 128 bytes will be read as four 128-byte messages, and not as one 512-byte message, as would happen with byte-mode pipes. Named pipes also exist and have the same two modes as regular pipes. Named pipes can also be used over a network; regular pipes cannot.

Mailslots are a feature of NT not present in UNIX. They are similar to pipes in some ways, but not all. For one thing, they are one-way, whereas pipes are two-way. They can also be used over a network but do not provide guaranteed delivery. Finally, they allow the sending process to broadcast a message to many receivers, instead of to just one.

Sockets are like pipes, except that they normally connect processes on different machines. However, they can also be used to connect processes on the same machine. In general, there is usually little advantage to using a socket connection over a pipe or named pipe for intramachine communication.

Remote procedure calls are a way for process *A* to have process *B* call a procedure in *B*'s address space on *A*'s behalf and return the result to *A*. Various restrictions on the parameters exist. For example, it makes no sense to pass a pointer to a different process.

Finally, processes can share memory by mapping onto the same file at the same time. All writes done by one process then appear in the address spaces of the other processes. Using this mechanism, the shared buffer used in our producer-consumer example can be easily implemented.

Just as NT provides numerous interprocess communication mechanisms, it also provides numerous synchronization mechanisms, including semaphores, mutexes, critical sections, and events. All of these mechanisms work on threads, not processes, so that when a thread blocks on a semaphore, other threads in that process (if any) are not affected and can continue to run.

A semaphore is created using the CreateSemaphore API function, which can initialize it to a given value and define a maximum value as well. Semaphores are kernel objects and thus have security descriptors and handles. The handle for a semaphore can be duplicated using DuplicateHandle and passed to another process so that multiple processes can synchronize on the same semaphore. Calls for up and down are present, although they have the somewhat peculiar names of ReleaseSemaphore (up) and WaitForSingleObject (down). It is also possible to give WaitForSingleObject a timeout, so the calling thread can be released eventually, even if the semaphore remains at 0 (although timers reintroduce races).

Mutexes are also kernel objects used for synchronization, but simpler than semaphores because they do not have counters. They are essentially locks, with API functions for locking (WaitForSingleObject) and unlocking ReleaseMutex. Like semaphore handles, mutex handles can be duplicated and passed between processes so that threads in different processes can access the same mutex.

The third synchronization mechanism is based on **critical sections**, which are similar to mutexes, except local to the address space of the creating thread. Because critical sections are not kernel objects, they do not have handles or security descriptors and cannot be passed between processes. Locking and unlocking is done with EnterCriticalSection and LeaveCriticalSection, respectively. Because these API functions are performed entirely in user space, they are much faster than mutexes.

The last synchronization mechanism uses kernel objects called **events**. A thread can wait for an event to occur with WaitForSingleObject. A thread can release a single thread waiting on an event with SetEvent or it can release all threads waiting on an event with PulseEvent. Events come in several flavors and have a variety of options, too.

Events, mutexes, and semaphores can all be named and stored in the file system, like named pipes. Two or more processes can synchronize by opening the same event, mutex, or semaphore, rather than having one of them create the object and then make duplicate handles for the others, although the latter approach is certainly an option as well.

7

THE ASSEMBLY LANGUAGE LEVEL

	Programmer-years to produce the program	Program execution time in seconds
Assembly language	50	33
High-level language	10	100
Mixed approach before tuning		
Critical 10%	1	90
Other 90%	9	10
Total	10	100
Mixed approach after tuning		
Critical 10%	6	30
Other 90%	9	10
Total	15	40

Figure 7-1. Comparison of assembly language and high-level language programming, with and without tuning.

```
Label       Opcode   Operands        Comments
FORMULA:    MOV      EAX,I           ; register EAX = I
            ADD      EAX,J           ; register EAX = I + J
            MOV      N,EAX           ; N = I + J

I           DW       3               ; reserve 4 bytes initialized to 3
J           DW       4               ; reserve 4 bytes initialized to 4
N           DW       0               ; reserve 4 bytes initialized to 0
```

(a)

```
Label       Opcode   Operands        Comments
FORMULA     MOVE.L   I, D0           ; register D0 = I
            ADD.L    J, D0           ; register D0 = I + J
            MOVE.L   D0, N           ; N = I + J

I           DC.L     3               ; reserve 4 bytes initialized to 3
J           DC.L     4               ; reserve 4 bytes initialized to 4
N           DC.L     0               ; reserve 4 bytes initialized to 0
```

(b)

```
Label       Opcode   Operands            Comments
FORMULA:    SETHI    %HI(I),%R1          ! R1 = high-order bits of the address of I
            LD       [%R1+%LO(I)],%R1    ! R1 = I
            SETHI    %HI(J),%R2          ! R2 = high-order bits of the address of J
            LD       [%R2+%LO(J)],%R2    ! R2 = J
            NOP                          ! wait for J to arrive from memory
            ADD      %R1,%R2,%R2         ! R2 = R1 + R2
            SETHI    %HI(N),%R1          ! R1 = high-order bits of the address of N
            ST       %R2,[%R1+%LO(N)]

I:          .WORD 3                      ! reserve 4 bytes initialized to 3
J:          .WORD 4                      ! reserve 4 bytes initialized to 4
N:          .WORD 0                      ! reserve 4 bytes initialized to 0
```

(c)

Figure 7-2. Computation of $N = I + J$. (a) Pentium II. (b) Motorola 680x0. (c) SPARC.

Pseudoinstr	Meaning
SEGMENT	Start a new segment (text, data, etc.) with certain attributes
ENDS	End the current segment
ALIGN	Control the alignment of the next instruction or data
EQU	Define a new symbol equal to a given expression
DB	Allocate storage for one or more (initialized) bytes
DD	Allocate storage for one or more (initialized) 16-bit halfwords
DW	Allocate storage for one or more (initialized) 32-bit words
DQ	Allocate storage for one or more (initialized) 64-bit double words
PROC	Start a procedure
ENDP	End a procedure
MACRO	Start a macro definition
ENDM	End a macro definition
PUBLIC	Export a name defined in this module
EXTERN	Import a name from another module
INCLUDE	Fetch and include another file
IF	Start conditional assembly based on a given expression
ELSE	Start conditional assembly if the IF condition above was false
ENDIF	End conditional assembly
COMMENT	Define a new start-of-comment character
PAGE	Generate a page break in the listing
END	Terminate the assembly program

Figure 7-3. Some of the pseudoinstructions available in the Pentium II assembler (MASM).

```
MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX

MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX
```

(a)

```
SWAP    MACRO
        MOV EAX,P
        MOV EBX,Q
        MOV Q,EAX
        MOV P,EBX
        ENDM

        SWAP

        SWAP
```

(b)

Figure 7-4. Assembly language code for interchanging P and Q twice. (a) Without a macro. (b) With a macro.

Item	Macro call	Procedure call
When is the call made?	During assembly	During execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	1

Figure 7-5. Comparison of macro calls with procedure calls.

```
MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX

MOV     EAX,R
MOV     EBX,S
MOV     S,EAX
MOV     R,EBX
```

(a)

```
CHANGE  MACRO P1, P2
        MOV EAX,P1
        MOV EBX,P2
        MOV P2,EAX
        MOV P1,EBX
        ENDM

        CHANGE P, Q

        CHANGE R, S
```

(b)

Figure 7-6. Nearly identical sequences of statements. (a) Without a macro. (b) With a macro.

Label	Opcode	Operands	Comments	Length	ILC
MARIA:	MOV	EAX,I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	branch to DONE	5	129

Figure 7-7. The instruction location counter (ILC) keeps track of the address where the instructions will be loaded in memory. In this example, the statements prior to MARIA occupy 100 bytes.

Symbol	Value	Other information
MARIA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

Figure 7-8. A symbol table for the program of Fig. 7-7.

Opcode	First operand	Second operand	Hexadecimal opcode	Instruction length	Instruction class
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

Figure 7-9. A few excerpts from the opcode table for a Pentium II assembler.

```
public static void pass_one( )  {
   // This procedure is an outline of pass one of a simple assembler.
   boolean more_input = true;       // flag that stops pass one
   String line, symbol, literal, opcode;     // fields of the instruction
   int location_counter, length, value, type;  // misc. variables
   final int END_STATEMENT = –2;  // signals end of input

   location_counter = 0;            // assemble first instruction at 0
   initialize_tables( );            // general initialization

   while (more_input) {             // more_input set to false by END
      line = read_next_line( );     // get a line of input
      length = 0;                   // # bytes in the instruction
      type = 0;                     // which type (format) is the instruction

      if (line_is_not_comment(line)) {
         symbol = check_for_symbol(line);    // is this line labeled?
         if (symbol != null)         // if it is, record symbol and value
            enter_new_symbol(symbol, location_counter);
         literal = check_for_literal(line);    // does line contain a literal?
         if (literal != null)        // if it does, enter it in table
            enter_new_literal(literal);

         // Now determine the opcode type.  –1 means illegal opcode.
         opcode = extract_opcode(line);   // locate opcode mnemonic
         type = search_opcode_table(opcode);    // find format, e.g. OP REG1,REG2
         if (type < 0)               // if not an opcode, is it a pseudoinstruction?
            type = search_pseudo_table(opcode);
         switch(type) {              // determine the length of this instruction
            case 1: length = get_length_of_type1(line);  break;
            case 2: length = get_length_of_type2(line);  break;
            // other cases here
         }
      }

      write_temp_file(type, opcode, length, line);// useful info for pass two
      location_counter = location_counter + length;// update loc_ctr
      if (type == END_STATEMENT) {    // are we done with input?
         more_input = false;         // if so, perform housekeeping tasks
         rewind_temp_for_pass_two( );    // like rewinding the temp file
         sort_literal_table( );      // and sorting the literal table
         remove_redundant_literals( ); // and removing duplicates from it
      }
   }
}
```

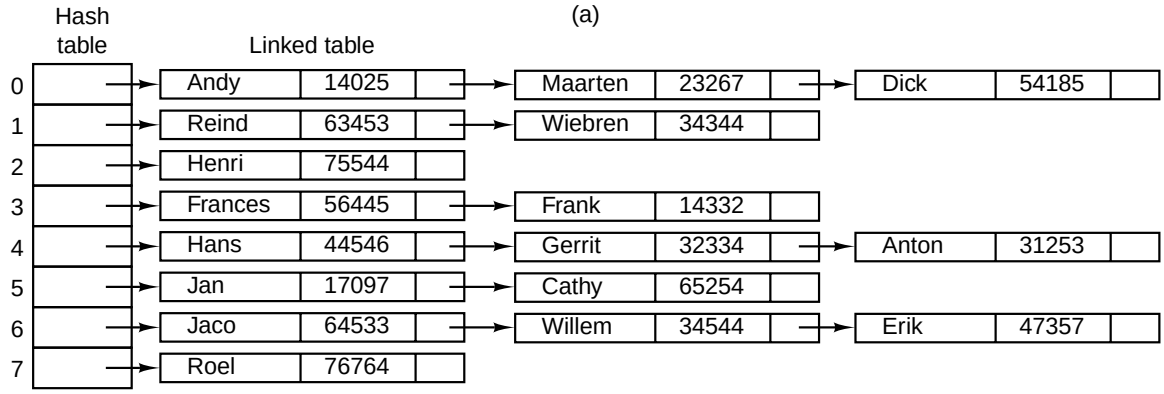
Figure 7-10. Pass one of a simple assembler.

```
public static void pass_two( )  {
   // This procedure is an outline of pass two of a simple assembler.
   boolean more_input = true; // flag that stops pass one
   String line, opcode;          // fields of the instruction
   int location_counter, length, type;   // misc. variables
   final int END_STATEMENT = –2;   // signals end of input
   final int MAX_CODE = 16;// max bytes of code per instruction
   byte code[ ] = new byte[MAX_CODE]; // holds generated code per instruction

   location_counter = 0;        // assemble first instruction at 0

   while (more_input) {         // more_input set to false by END
      type = read_type( );       // get type field of next line
      opcode = read_opcode( );  // get opcode field of next line
      length = read_length( ); // get length field of next line
      line = read_line( );         // get the actual line of input

      if (type != 0) {             // type 0 is for comment lines
         switch(type) {            // generate the output code
            case 1: eval_type1(opcode, length, line, code);  break;
            case 2: eval_type2(opcode, length, line, code);  break;
            // other cases here
         }
      }

      write_output(code);       // write the binary code
      write_listing(code, line);// print one line on the listing
      location_counter = location_counter + length;// update loc_ctr
      if (type == END_STATEMENT) {// are we done with input?
         more_input = false;    // if so, perform housekeeping tasks
         finish_up( );              // odds and ends
      }
   }
}
```

Figure 7-11. Pass two of a simple assembler.

Andy	14025	0
Anton	31253	4
Cathy	65254	5
Dick	54185	0
Erik	47357	6
Frances	56445	3
Frank	14332	3
Gerrit	32334	4
Hans	44546	4
Henri	75544	2
Jan	17097	5
Jaco	64533	6
Maarten	23267	0
Reind	63453	1
Roel	76764	7
Willem	34544	6
Wiebren	34344	1

(a)



(b)

Figure 7-12. Hash coding. (a) Symbols, values, and the hash codes derived from the symbols. (b) Eight-entry hash table with linked lists of symbols and values.

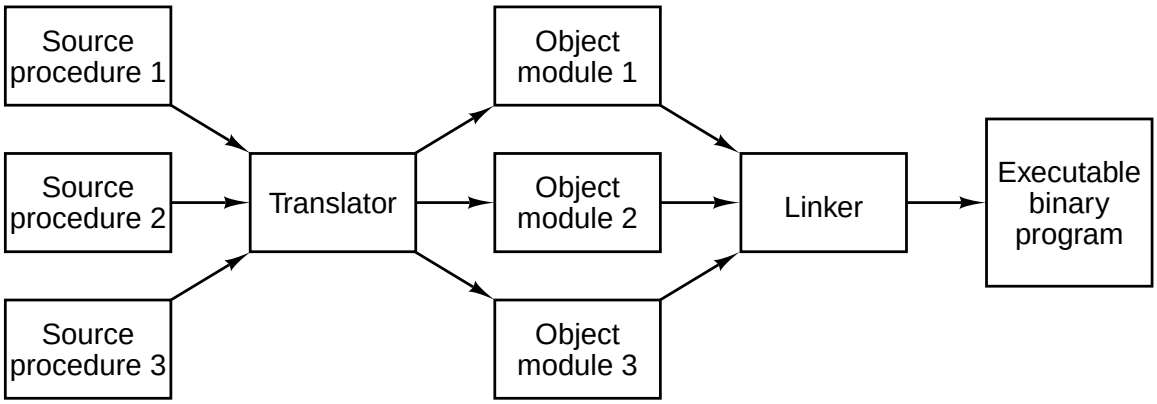


Figure 7-13. Generation of an executable binary program from a collection of independently translated source procedures requires using a linker.

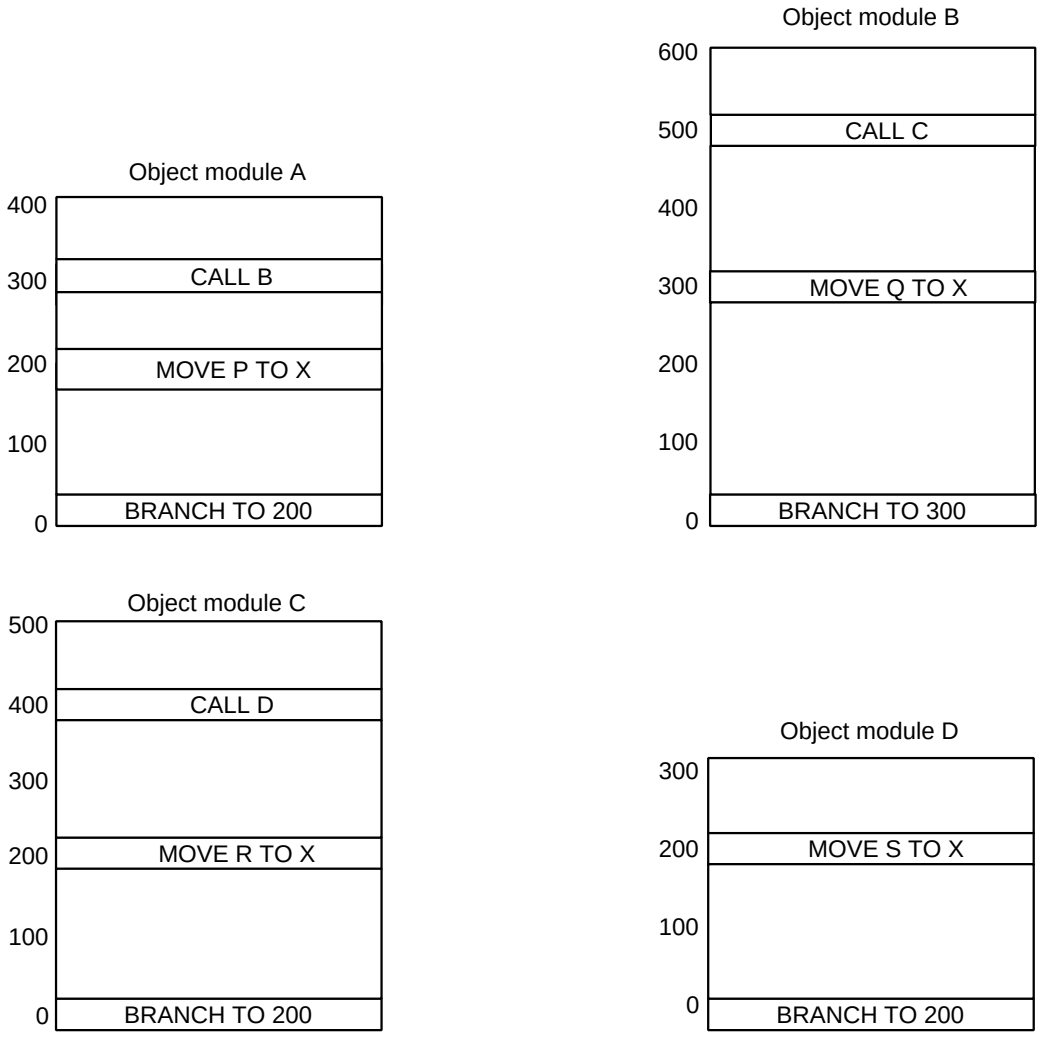


Figure 7-14. Each module has its own address space, starting at 0.

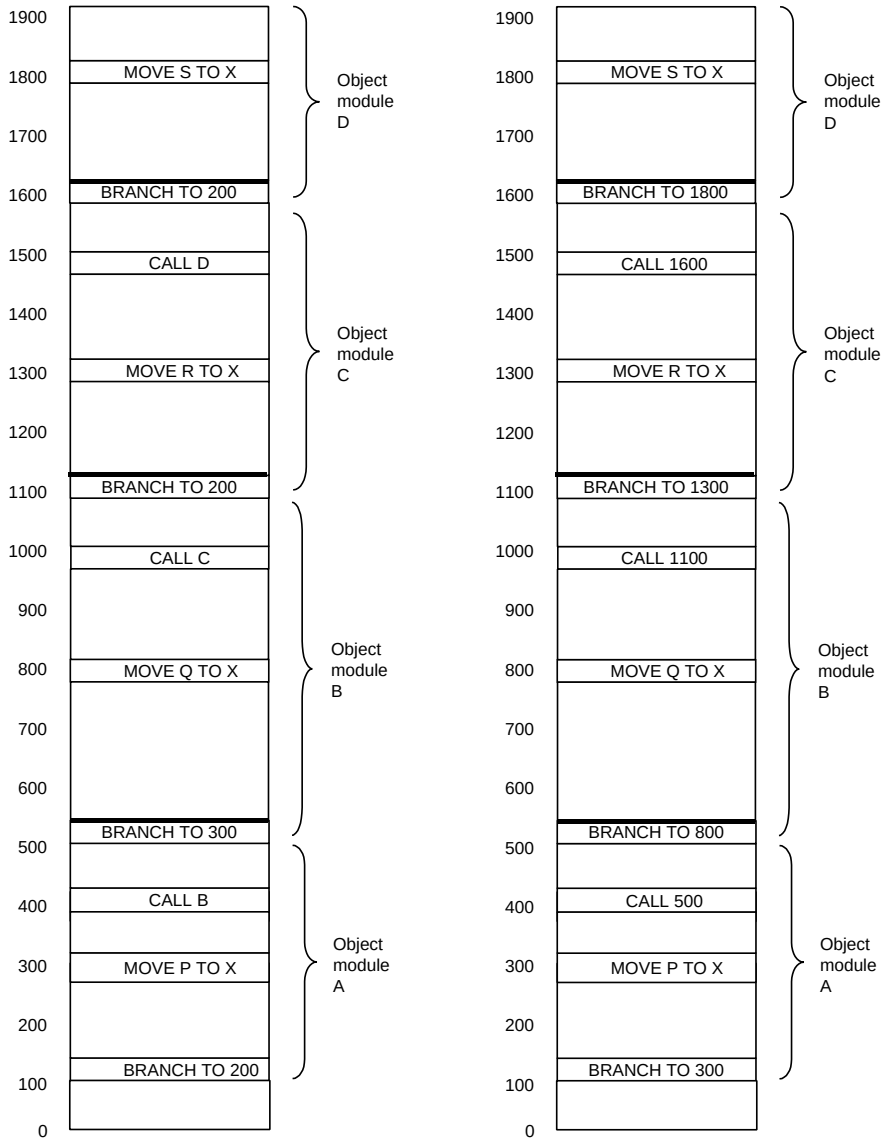


Figure 7-15. (a) The object modules of Fig. 7-14 after being positioned in the binary image but before being relocated and linked. (b) The same object modules after linking and after relocation has been performed. Together they form an executable binary program, ready to run.

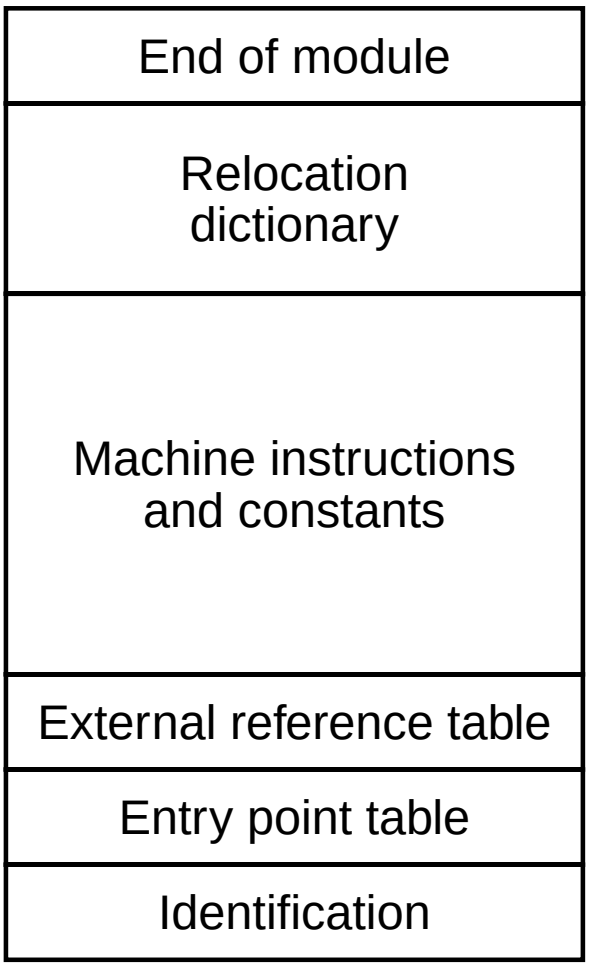


Figure 7-16. The internal structure of an object module produced by a translator.

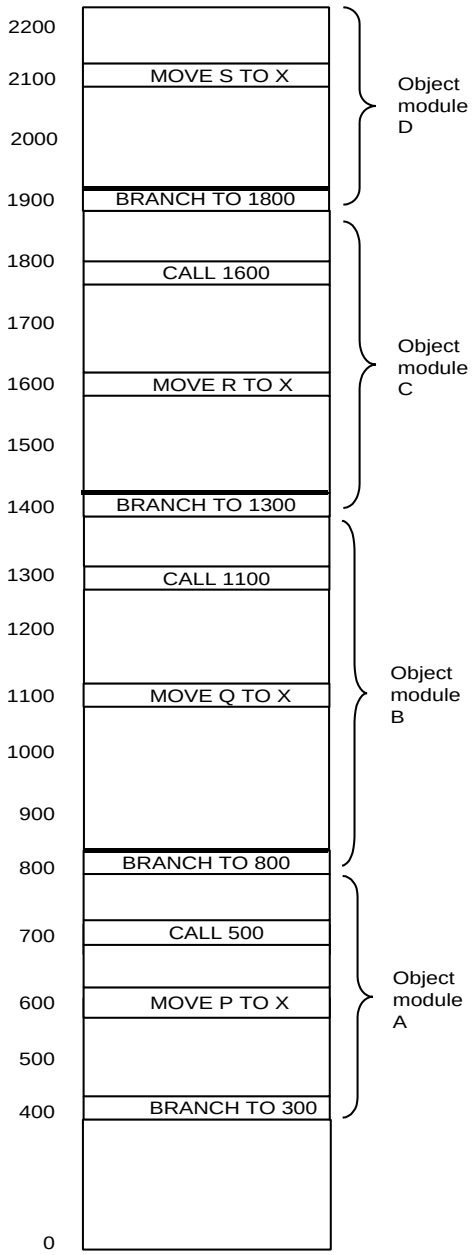
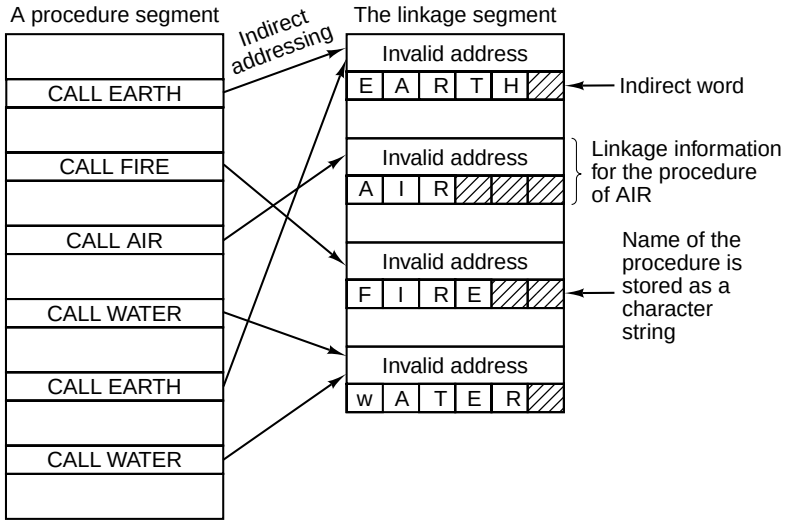
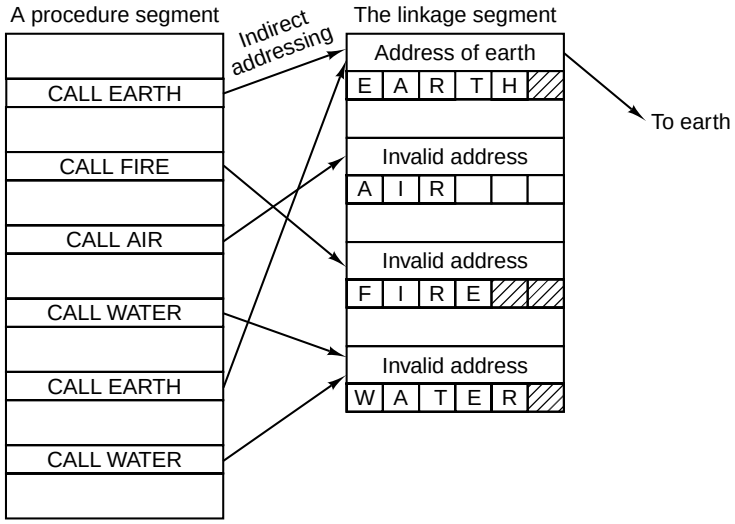


Figure 7-17. The relocated binary program of Fig. 7-15(b) moved up 300 addresses. Many instructions now refer to an incorrect memory address.



(a)



(b)

Figure 7-18. Dynamic linking. (a) Before *EARTH* is called. (b) After *EARTH* has been called and linked.

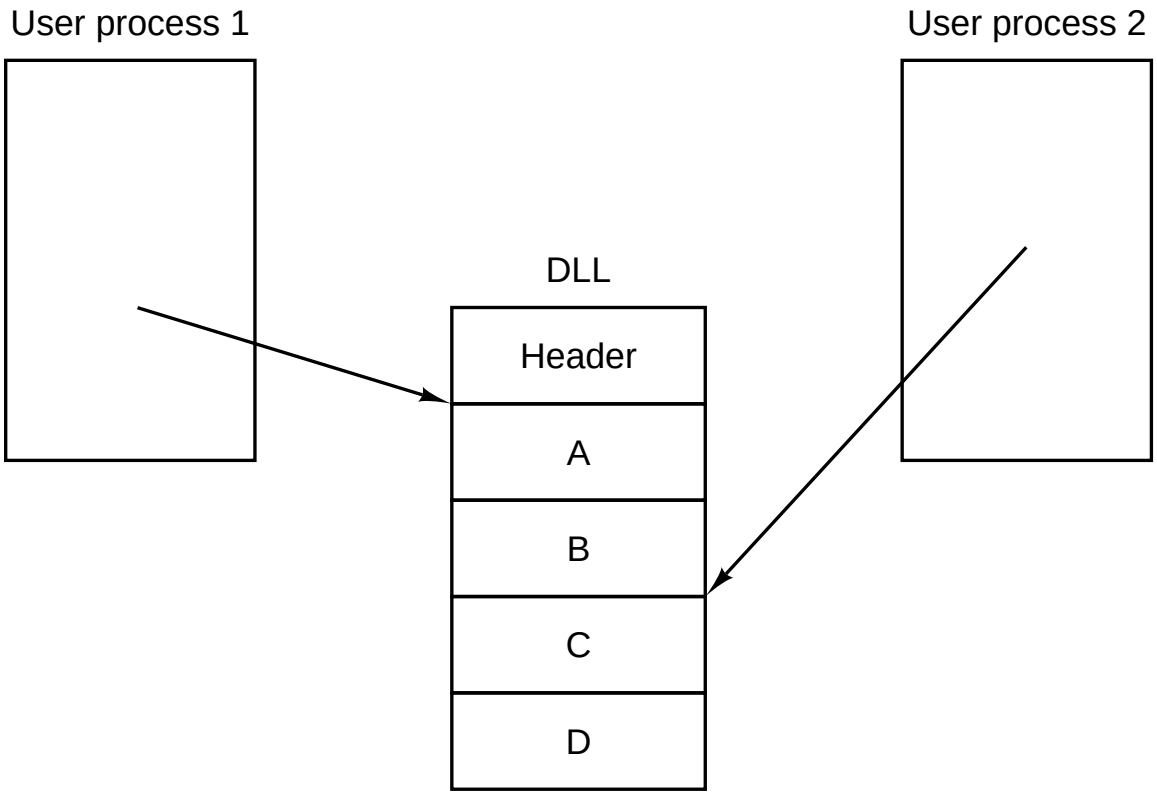


Figure 7-19. Use of a DLL file by two processes.

7.4 LINKING AND LOADING

Most programs consist of more than one procedure. Compilers and assemblers generally translate one procedure at a time and put the translated output on disk. Before the program can be run, all the translated procedures must be found and linked together properly. If virtual memory is not available, the linked program must be explicitly loaded into main memory as well. Programs that perform these functions are called by various names, including **linker**, **linking loader**, and **linkage editor**. The complete translation of a source program requires two steps, as shown in Fig. 7-1:

1. Compilation or assembly of the source procedures.
2. Linking of the object modules.

The first step is performed by the compiler or assembler and the second one is performed by the linker.

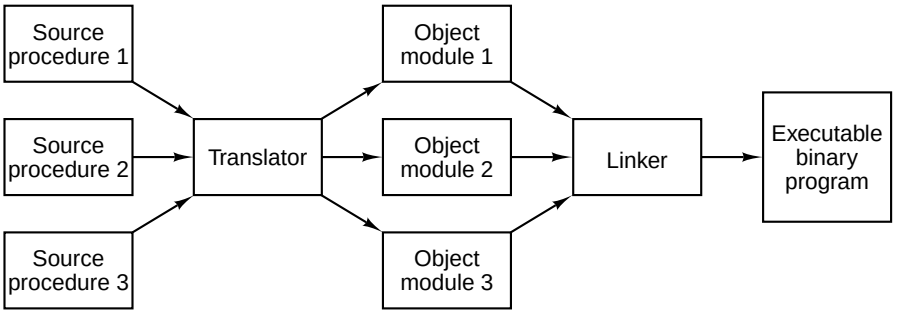


Figure 7-1. Generation of an executable binary program from a collection of independently translated source procedures requires using a linker.

The translation from source procedure to object module represents a change of level because the source language and target language have different instructions and notation. The linking process, however, does not represent a change of level, since both the linker's input and the linker's output are programs for the same virtual machine. The linker's function is to collect procedures translated separately and link them together to be run as a unit called an **executable binary program**. On MS-DOS, Windows 95/98, and NT the object modules have extension *.obj* and the executable binary programs have extension *.exe*. On UNIX, the object modules have extension *.o*; executable binary programs have no extension.

Compilers and assemblers translate each source procedure as a separate entity for a good reason. If a compiler or assembler were to read a series of source procedures and directly produce a ready-to-run machine language program, changing one statement in one source procedure would require that all the source procedures be retranslated.

If the separate-object-module technique of Fig. 7-1 is used, it is only necessary to retranslate the modified procedure and not the unchanged ones, although it is necessary to relink all the object modules again. Linking is usually much faster than translating, however; thus the two-step process of translating and linking can save a great deal of time during the development of a program. This gain is especially important for programs with hundreds or thousands of modules.

7.4.1 Tasks Performed by the Linker

At the start of pass one of the assembly process, the instruction location counter is set to 0. This step is equivalent to assuming that the object module will be located at (virtual) address 0 during execution. Figure 7-2 shows four object modules for a generic machine. In this example, each module begins with a BRANCH instruction to a MOVE instruction within the module.

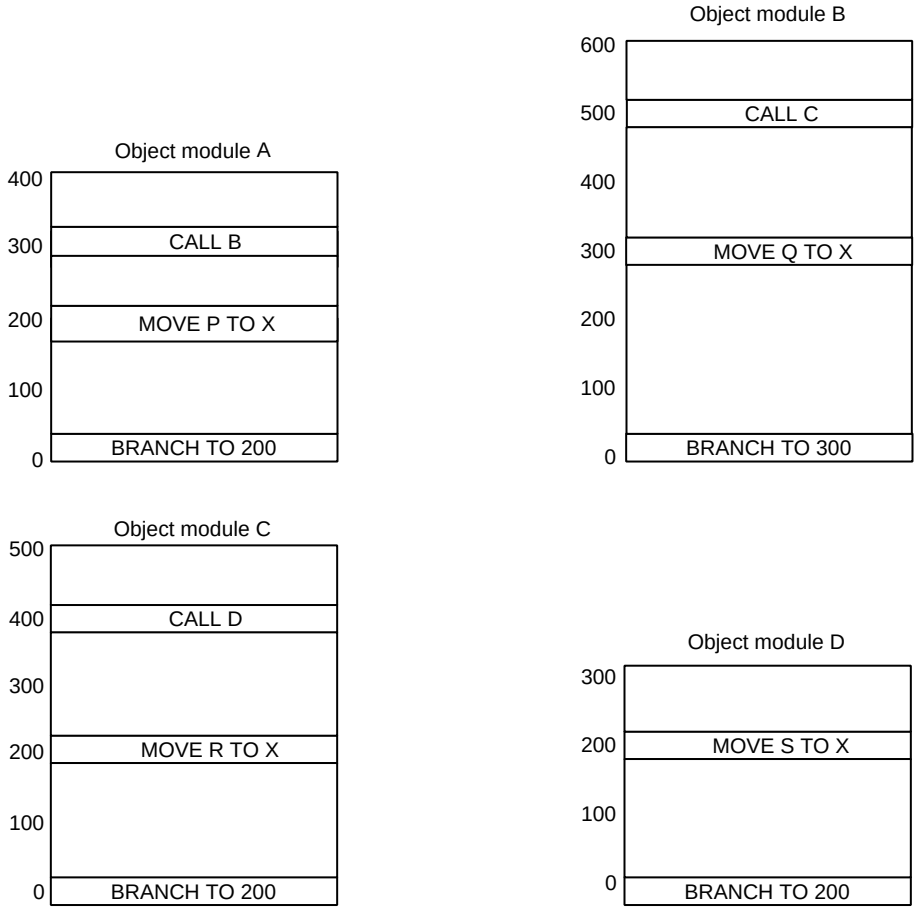


Figure 7-2. Each module has its own address space, starting at 0.

In order to run the program, the linker brings the object modules into main memory to form the image of the executable binary program, as shown in Fig. 7-3(a). The idea is to make an exact image of the executable program's virtual address space inside the linker and position all the object modules at their correct locations. If there is not enough (virtual) memory to form the image, a disk file can be used. Typically, a small section of memory starting at address zero is used for interrupt vectors, communication with the operating system, catching uninitialized pointers, or other purposes, so programs often start above 0. In this figure we have (arbitrarily) started programs at address 100.

The program of Fig. 7-3(a), although loaded into the image of the executable binary file, is not yet ready for execution. Consider what would happen if execution began with the instruction at the beginning of module *A*. The program would not branch to the MOVE instruction as it should, because that instruction is now at 300. In fact, all memory reference instructions will fail for the same reason.

This problem, called the **relocation problem**, occurs because each object module in Fig. 7-2 represents a separate address space. On a machine with a segmented address space, such as the Pentium II, theoretically each object module could have its own address space by being placed in its own segment. However, OS/2 is the only operating system for the Pentium II that supports this concept. All versions of Windows and UNIX support only one linear address space, so all the object modules must be merged together into a single address space.

Furthermore, the procedure call instructions in Fig. 7-3(a) will not work either. At address 400, the programmer had intended to call object module *B*, but because each procedure is translated by itself, the assembler has no way of knowing what address to insert into the CALL B instruction. The address of object module *B* is not known until linking time. This problem is called the **external reference** problem. Both of these problems can be solved by the linker.

The linker merges the separate address spaces of the object modules into a single linear address space in the following steps:

1. It constructs a table of all the object modules and their lengths.
2. Based on this table, it assigns a starting address to each object module.
3. It finds all the instructions that reference memory adds to each a **relocation constant** equal to the starting address of its module.
4. It finds all the instructions that reference other procedures and inserts the address of these procedures in place.

The object module table constructed in step 1 is shown for the modules of Fig. 7-3 below. It gives the name, length, and starting address of each module.

Module	Length	Starting address
A	400	100
B	600	500

C	500	1100
D	300	1600

Figure 7-3(b) shows how the address space of Fig. 7-3(a) looks after the linker has performed these steps.

7.4.2 Structure of an Object Module

Object modules often contain six parts, as shown in Fig. 7-4. The first part contains the name of the module, certain information needed by the linker, such as the lengths of the various parts of the module, and sometimes the assembly date.

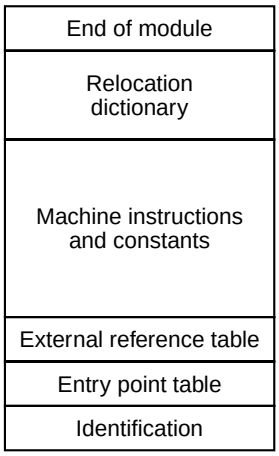


Figure 7-4. The internal structure of an object module produced by a translator.

The second part of the object module is a list of the symbols defined in the module that other modules may reference, together with their values. For example, if the module consists of a procedure named *bigbug*, the entry point table will contain the character string ‘‘bigbug’’ followed by the address to which it corresponds. The assembly language programmer indicates which symbols are to be declared as **entry points** by using a pseudoinstruction such as PUBLIC in Fig. 7-0.

The third part of the object module consists of a list of the symbols that are used in the module but which defined in other modules, along with a list of which machine instructions use which symbols. The linker needs the latter list in order to be able to insert the correct addresses into the instructions that use external symbols. A procedure can call other independently translated procedures by declaring the names of the called procedures to be external. The assembly language programmer indicates which symbols are to be declared as **external symbols** by using a pseudoinstruction such as EXTERN in Fig. 7-0. On some computers entry points and external references are combined into one table.

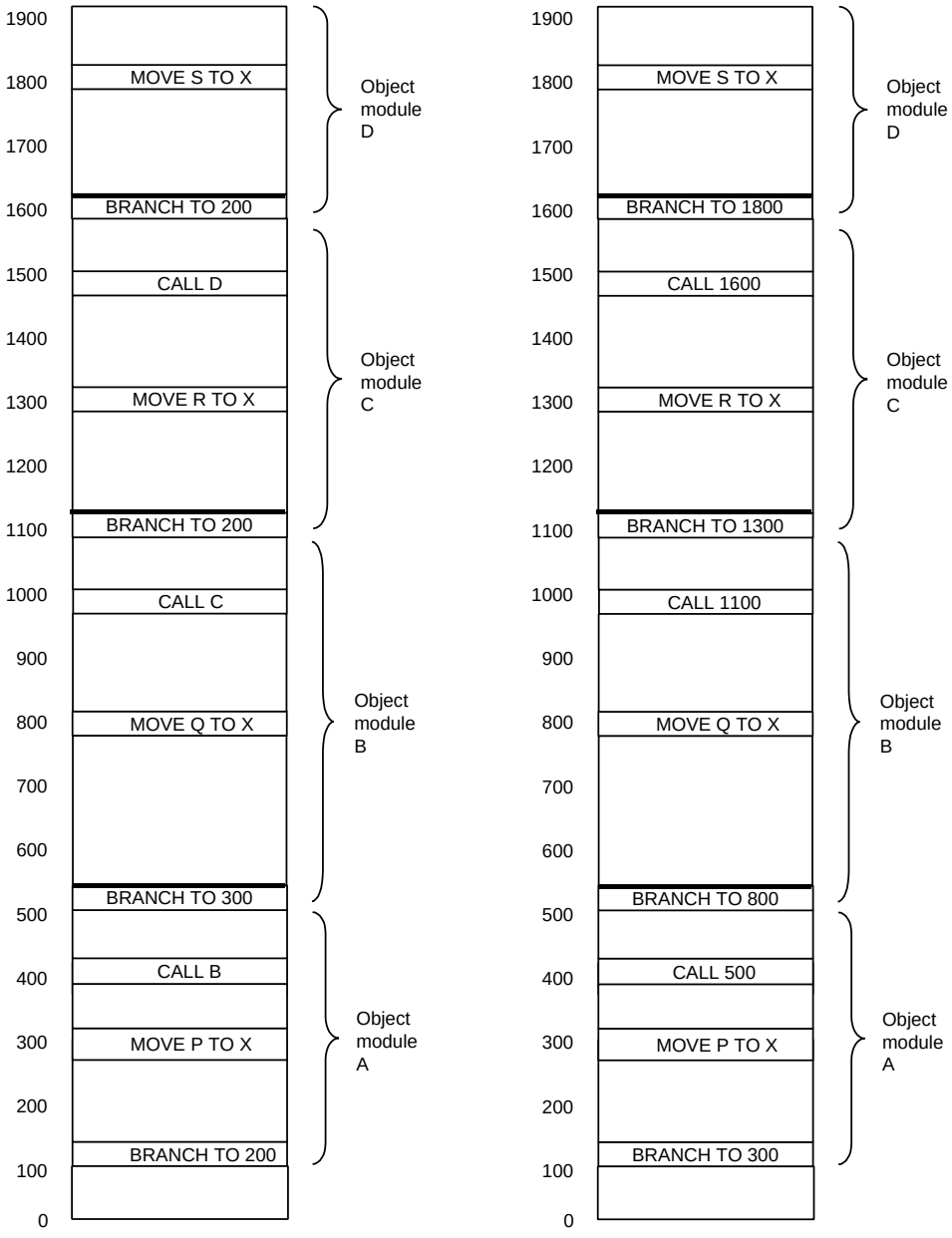


Figure 7-3. (a) The object modules of Fig. 7-2 after being positioned in the binary image but before being relocated and linked. (b) The same object modules after linking and after relocation has been performed. Together they form an executable binary program, ready to run.

The fourth part of the object module is the assembled code and constants. This part of the object module is the only one that will be loaded into memory to be executed. The other five parts will be used by the linker and then discarded before execution begins.

The fifth part of the object module is the relocation dictionary. As shown in Fig. 7-3, instructions that contain memory addresses must have a relocation constant added. Since the linker has no way of telling by inspection which of the data words in part 4 contain machine instructions and which contain constants, information about which addresses are to be relocated is provided in this table. The information may take the form of a bit table, with 1 bit per potentially relocatable address, or an explicit list of addresses to be relocated.

The sixth part is an end-of-module indication, sometimes a checksum to catch errors made while reading the module, and the address at which to begin execution.

Most linkers require two passes. On pass one the linker reads all the object modules and builds up a table of module names and lengths, and a global symbol table consisting of all entry points and external references. On pass two the object modules are read, relocated, and linked one module at a time.

7.4.3 Binding Time and Dynamic Relocation

In a multiprogramming system, a program can be read into main memory, run for a little while, written to disk, and then read back into main memory to be run again. In a large system, with many programs, it is difficult to ensure that a program is read back into the same locations every time.

Figure 7-5 shows what would happen if the already relocated program of Fig. 7-3(b) were reloaded at address 400 instead of address 100 where the linker put it originally. All the memory addresses are incorrect; moreover, the relocation information has long since been discarded. Even if the relocation information were still available, the cost of having to relocate all the addresses every time the program was swapped would be too high.

The problem of moving programs that have already been linked and relocated is intimately related to the time at which the final binding of symbolic names onto absolute physical memory addresses is completed. When a program is written it contains symbolic names for memory addresses, for example, `BR L`. The time at which the actual main memory address corresponding to *L* is determined is called the **binding time**. At least six possibilities for the binding time exist:

1. When the program is written.
2. When the program is translated.
3. When the program is linked but before it is loaded.

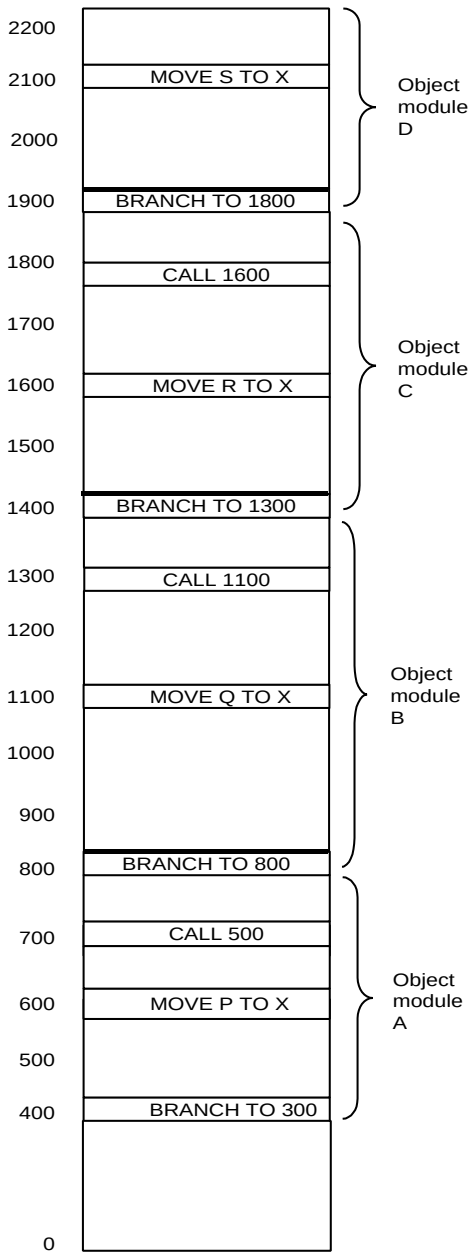


Figure 7-5. The relocated binary program of Fig. 7-3(b) moved up 300 addresses. Many instructions now refer to an incorrect memory address.

4. When the program is loaded.
5. When a base register used for addressing is loaded.
6. When the instruction containing the address is executed.

If an instruction containing a memory address is moved after binding, it will be incorrect (assuming that the object referred to has also been moved). If the translator produces an executable binary as output, the binding has occurred at translation time, and the program must be run at the address the translator expected it to be run at. The linking method described in the preceding section binds symbolic names to absolute addresses during linking, which is why moving programs after linking fails, as shown in Fig. 7-5.

Two related issues are involved here. First, there is the question of when symbolic names are bound to virtual addresses. Second, there is a question of when virtual addresses are bound to physical addresses. Only when both operations have taken place is binding complete. When the linker merges the separate address spaces of the object modules into a single linear address space, it is, in fact, creating a virtual address space. The relocation and linking serve to bind symbolic names onto specific virtual addresses. This observation is true whether or not virtual memory is being used.

Assume for the moment that the address space of Fig. 7-3(b) were paged. It is clear that the virtual addresses corresponding to the symbolic names *A*, *B*, *C*, and *D* have already been determined, even though their physical main memory addresses will depend on the contents of the page table at the time they are used. An executable binary program is really a binding of symbolic names onto virtual addresses.

Any mechanism that allows the mapping of virtual addresses onto physical main memory addresses to be changed easily will facilitate moving programs around in main memory, even after they have been bound to a virtual address space. One such mechanism is paging. After a program has been moved in main memory, only its page table need be changed, not the program itself.

A second mechanism is the use of a runtime relocation register. The CDC 6600 and its successors had such a register. On machines using this relocation technique, the register always points to the physical memory address of the start of the current program. All memory addresses have the relocation register added to them by the hardware before being sent to the memory. The entire relocation process is transparent to the user programs. They do not even know that it is occurring. When a program is moved, the operating system must update the relocation register. This mechanism is less general than paging because the entire program must be moved as a unit (unless there are separate code and data relocation registers, as on the Intel 8088, in which case it has to be moved as two units).

A third mechanism is possible on machines that can refer to memory relative to the program counter. Whenever a program is moved in main memory only the

program counter need be updated. A program, all of whose memory references are either relative to the program counter or absolute (e.g., to I/O device registers at absolute addresses) is said to be **position independent**. A position-independent procedure can be placed anywhere within the virtual address space without the need for relocation.

7.4.4 Dynamic Linking

The linking strategy discussed in Sec. 7.4.1 has the property that all procedures that a program might call are linked before the program can begin execution. On a computer with virtual memory, completing all linking before beginning execution does not take advantage of the full capabilities of the virtual memory. Many programs have procedures that are only called under unusual circumstances. For example, compilers have procedures for compiling rarely used statements, plus procedures for handling error conditions that seldom occur.

A more flexible way to link separately compiled procedures is to link each procedure at the time it is first called. This process is known as **dynamic linking**. It was pioneered by MULTICS whose implementation is in some ways still unsurpassed. In the next sections we will look at dynamic linking in several systems.

Dynamic Linking in MULTICS

In the MULTICS form of dynamic linking, associated with each program is a segment, called the **linkage segment**, which contains one block of information for each procedure that might be called. This block of information starts with a word reserved for the virtual address of the procedure and it is followed by the procedure name, which is stored as a character string.

When dynamic linking is being used, procedure calls in the source language are translated into instructions that indirectly address the first word of the corresponding linkage block, as shown in Fig. 7-6(a). The compiler fills this word with either an invalid address or a special bit pattern that forces a trap.

When a procedure in a different segment is called, the attempt to address the invalid word indirectly causes a trap to the dynamic linker. The linker then finds the character string in the word following the invalid address and searches the user's file directory for a compiled procedure with this name. That procedure is then assigned a virtual address, usually in its own private segment, and this virtual address overwrites the invalid address in the linkage segment, as indicated in Fig. 7-6(b). Next, the instruction causing the linkage fault is reexecuted, allowing the program to continue from the place it was before the trap.

All subsequent references to that procedure will be executed without causing a linkage fault, for the indirect word now contains a valid virtual address. Consequently, the dynamic linker is invoked only the first time a procedure is called and not thereafter.

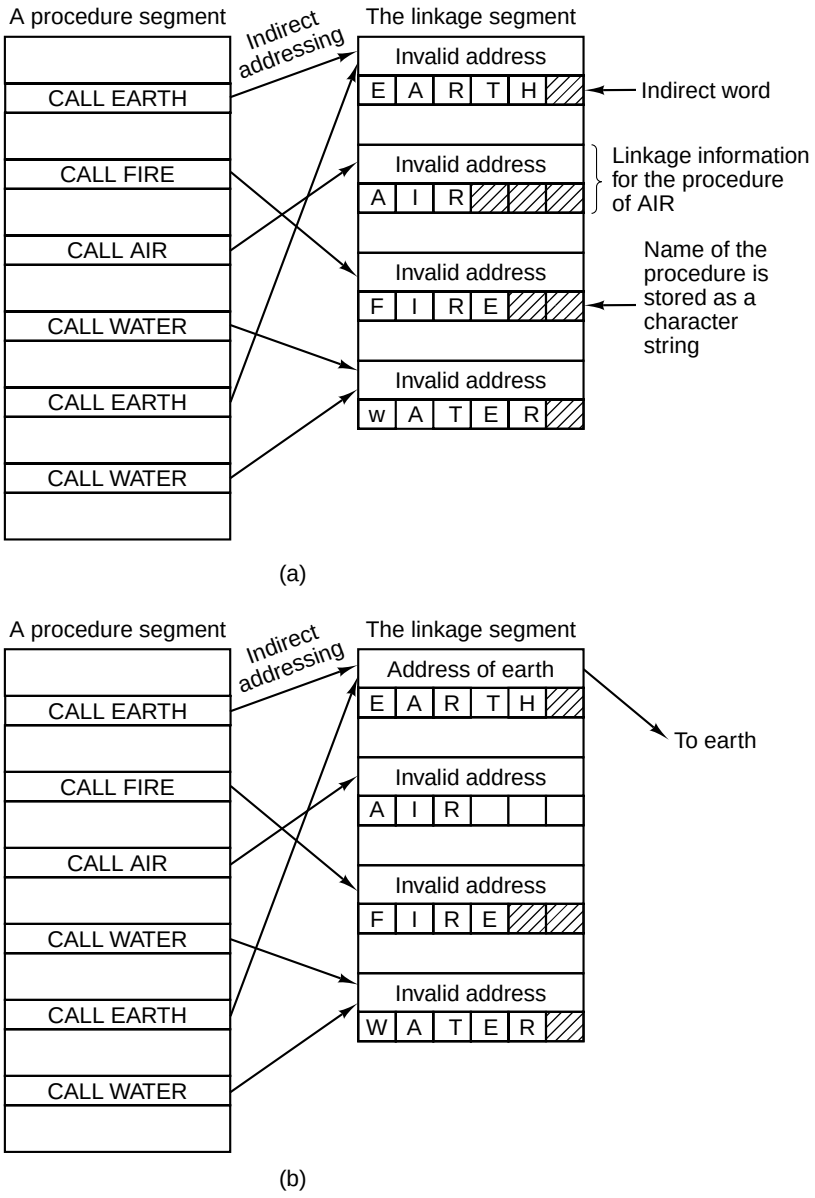


Figure 7-6. Dynamic linking. (a) Before *EARTH* is called. (b) After *EARTH* has been called and linked.

Dynamic Linking in Windows

All versions of the Windows operating system, including NT, support dynamic linking and rely heavily on it. Dynamic linking uses a special file format called a **DLL** (**Dynamic Link Library**). DLLs can contain procedures, data, or both. They are commonly used to allow two or more processes to share library procedures or data. Many DLLs have extension *.dll*, but other extensions are also in use, including *.drv* (for driver libraries) and *.fon* (for font libraries).

The most common form of a DLL is a library consisting of a collection of procedures that can be loaded into memory and accessed by multiple processes at the same time. Figure 7-7 illustrates two programs sharing a DLL file that contains four procedures, *A*, *B*, *C*, and *D*. Program 1 uses procedure *A*; program 2 uses procedure *C*, although they could equally well have used the same procedure.

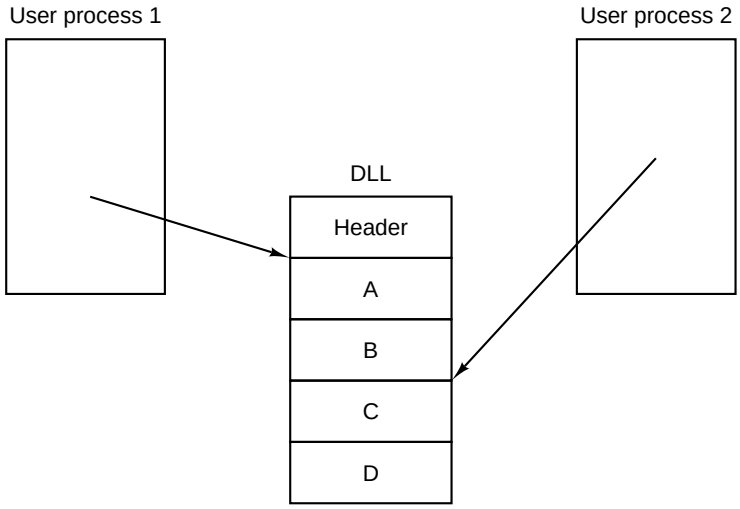


Figure 7-7. Use of a DLL file by two processes.

A DLL is constructed by the linker from a collection of input files. In fact, building a DLL file is very much like building an executable binary program, except that a special flag is given to the linker to tell it to make a DLL. DLLs are commonly constructed from collections of library procedures that are likely to be needed by multiple processes. The interface procedures to the Windows system call library and large graphics libraries are common examples of DLLs. The advantage of using DLLs is saving space in memory and on disk. If some common library were statically bound to each program using it, it would appear in many executable binaries on the disk and in memory, wasting space. With DLLs, each library only appears once on disk and once in memory.

In addition to saving space, this approach makes it easy to update library procedures, even after the programs using them have been compiled and linked. For

commercial software packages, where the users rarely have the source code, using DLLs means that the software vendor can fix bugs in the libraries by just distributing new DLL files over the Internet, without requiring any changes to the main program binaries.

The main difference between a DLL and an executable binary is that a DLL cannot be started and run on its own (because it has no main program). It also has different information in its header. In addition, the DLL as a whole has several extra procedures not related to the procedures in the library. For example, there is one procedure that is automatically called whenever a new process is bound to the DLL and another one that is automatically called whenever a process is unbound from it. These procedures can allocate and deallocate memory or manage other resources needed by the DLL.

There are two ways for a program to bind to a DLL. In the first way, called **implicit linking**, the user's program is statically linked with a special file called an **import library** that is generated by a utility program that extracts certain information from the DLL. The import library provides the glue that allows the user program to access the DLL. A user program can be linked with multiple import libraries. When a program using implicit linking is loaded into memory for execution, Windows examines it to see which DLLs it uses and checks to see if all of them are already in memory. Those that are not in memory are loaded immediately (but not necessarily in their entirety, since they are paged). Some changes are then made to the data structures in the import libraries so the called procedures can be located, somewhat analogous to the changes shown in Fig. 7-6. They also have to be mapped into the program's virtual address space. At this point, the user program is ready to run and can call the procedures in the DLLs as though they had been statically bound with it.

The alternative to implicit linking is (not surprisingly) **explicit linking**. This approach does not require import libraries and does not cause the DLLs to be loaded at the same time the user program is. Instead, the user program makes an explicit call at run time to bind to a DLL, then makes additional calls to get the addresses of procedures it needs. Once these have been found, it can call the procedures. When it is all done, it makes a final call to unbind from the DLL. When the last process unbinds from a DLL, the DLL can be removed from memory.

It is important to realize that a procedure in a DLL does not have any identity of its own (as a thread or process does). It runs in the caller's thread and uses the caller's stack for its local variables. It can have process-specific static data (as well as shared data) and otherwise behaves the same as a statically-linked procedure. The only essential difference is how the binding to it is performed.

Dynamic Linking in UNIX

The UNIX system has a mechanism essentially identical in essence to DLLs in Windows. It is called a **shared library**. Like a DLL file, a shared library is an archive file containing multiple procedures or data modules that are present in memory at run time and can be bound to multiple processes at the same time. The standard C library and much of the networking code are shared libraries.

UNIX supports only implicit linking, so a shared library consist of two parts: a **host library**, which is statically linked with the executable file, and a **target library**, which is called at run time. While the details differ, the concepts are essentially the same as with DLLs.

8

PARALLEL COMPUTER ARCHITECTURES

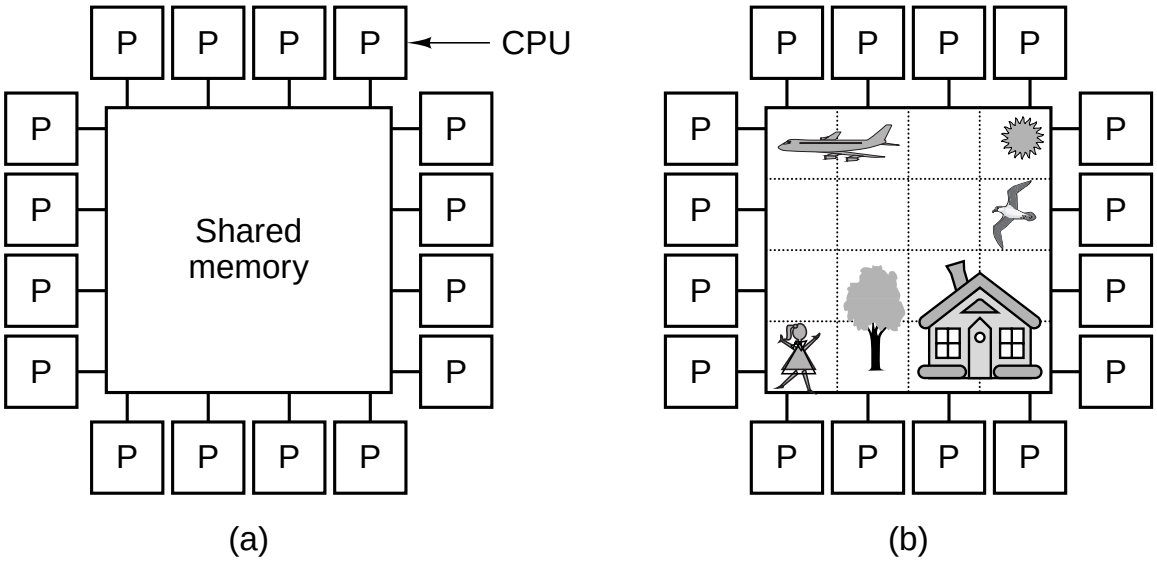


Figure 8-1. (a) A multiprocessor with 16 CPUs sharing a common memory. (b) An image partitioned into 16 sections, each being analyzed by a different CPU.

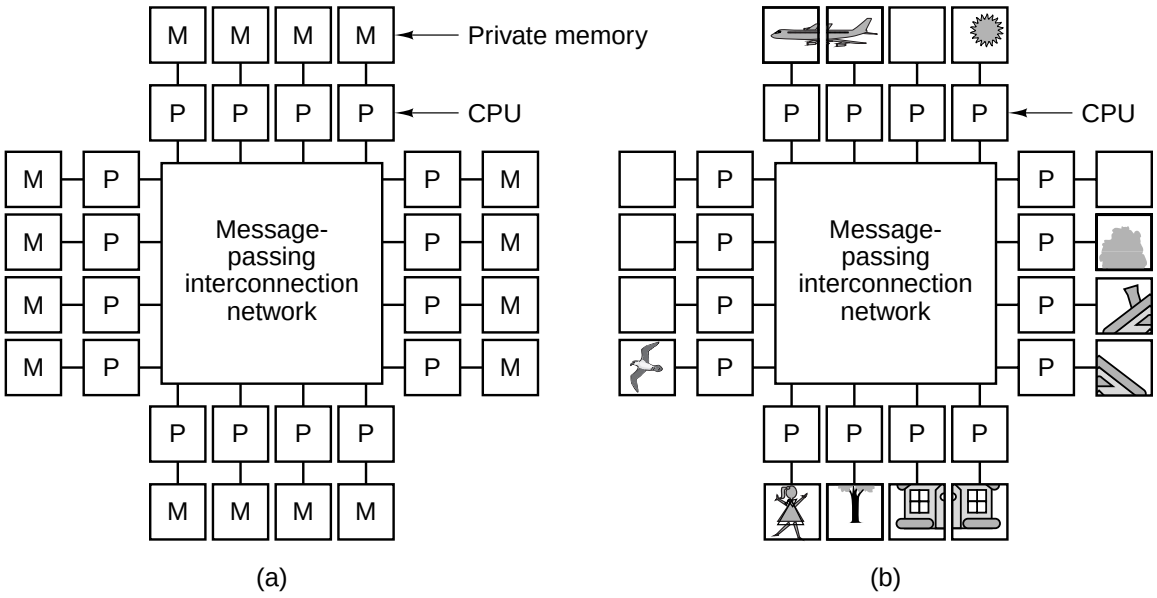


Figure 8-2. (a) A multicomputer with 16 CPUs, each with each own private memory. (b) The bit-map image of Fig. 8-1 split up among the 16 memories.

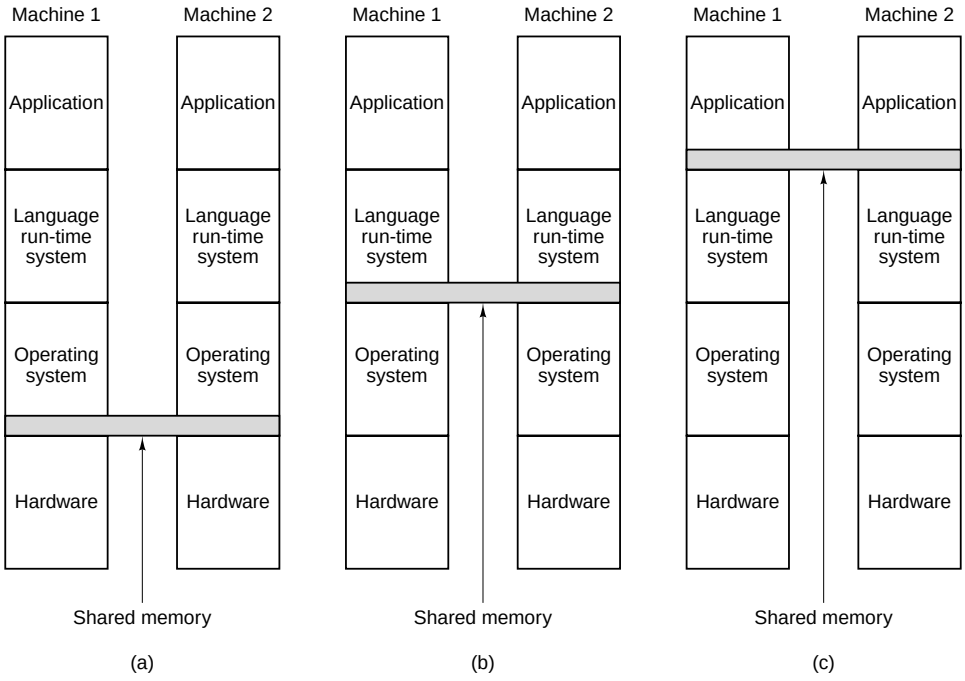


Figure 8-3. Various layers where shared memory can be implemented. (a) The hardware. (b) The operating system. (c) The language runtime system.

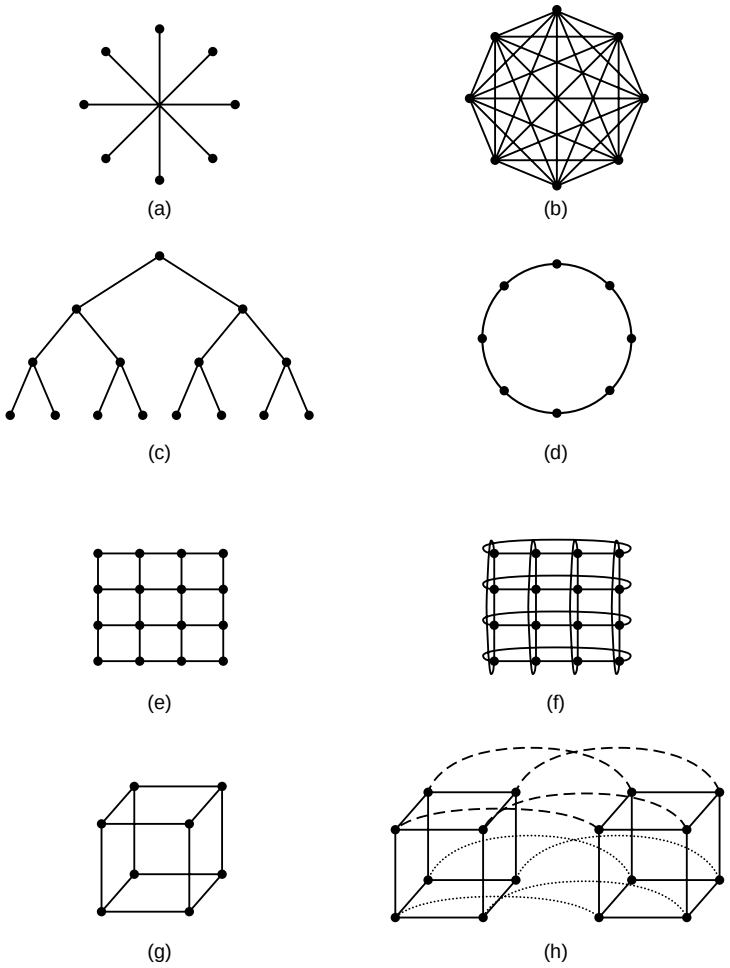


Figure 8-4. Various topologies. The heavy dots represent switches. The CPUs and memories are not shown. (a) A star. (b) A complete interconnect. (c) A tree. (d) A ring. (e) A grid. (f) A double torus. (g) A cube. (h) A 4D hypercube.

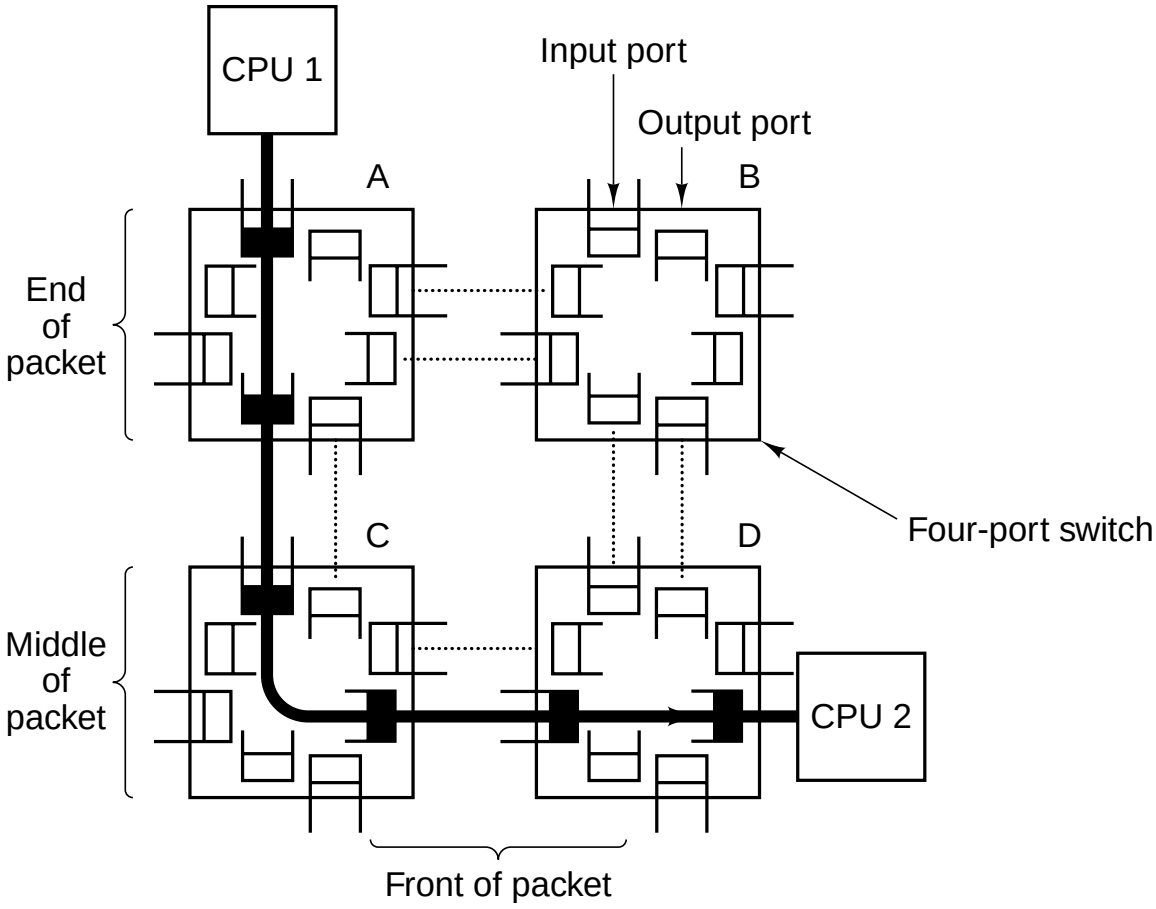


Figure 8-5. An interconnection network in the form of a four-switch square grid. Only two of the CPUs are shown.

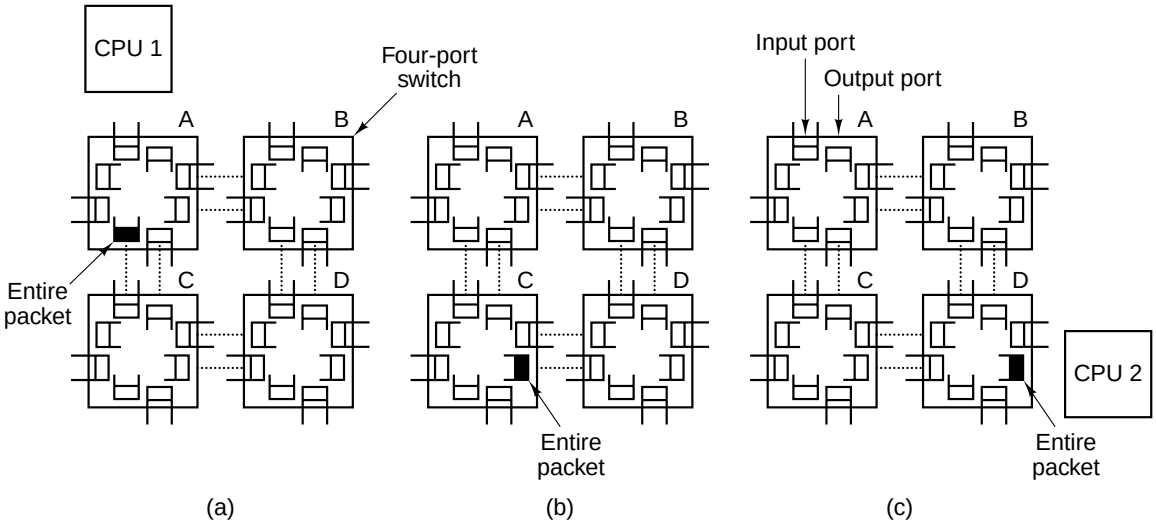


Figure 8-6. Store-and-forward packet switching.

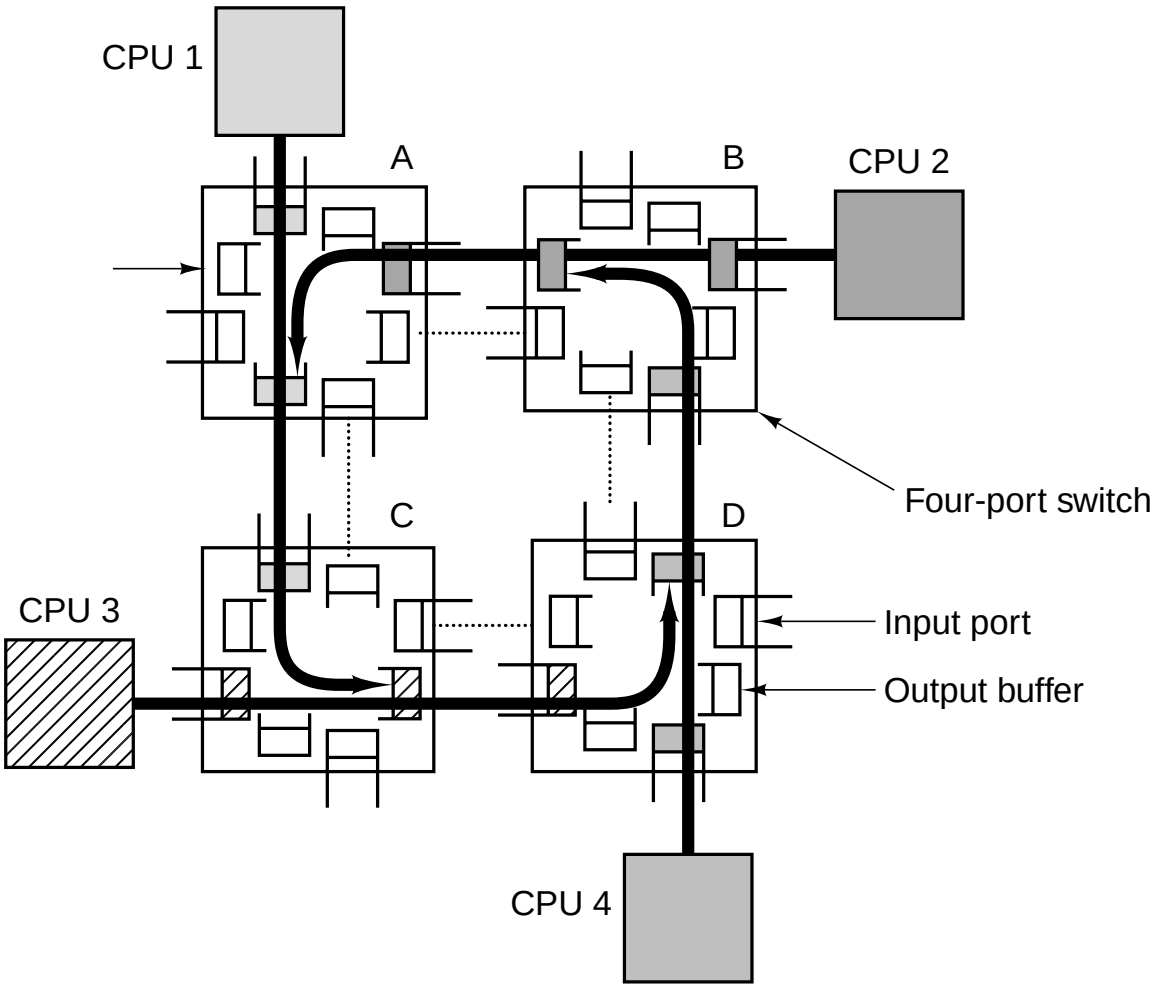


Figure 8-7. Deadlock in a circuit-switched interconnection network.

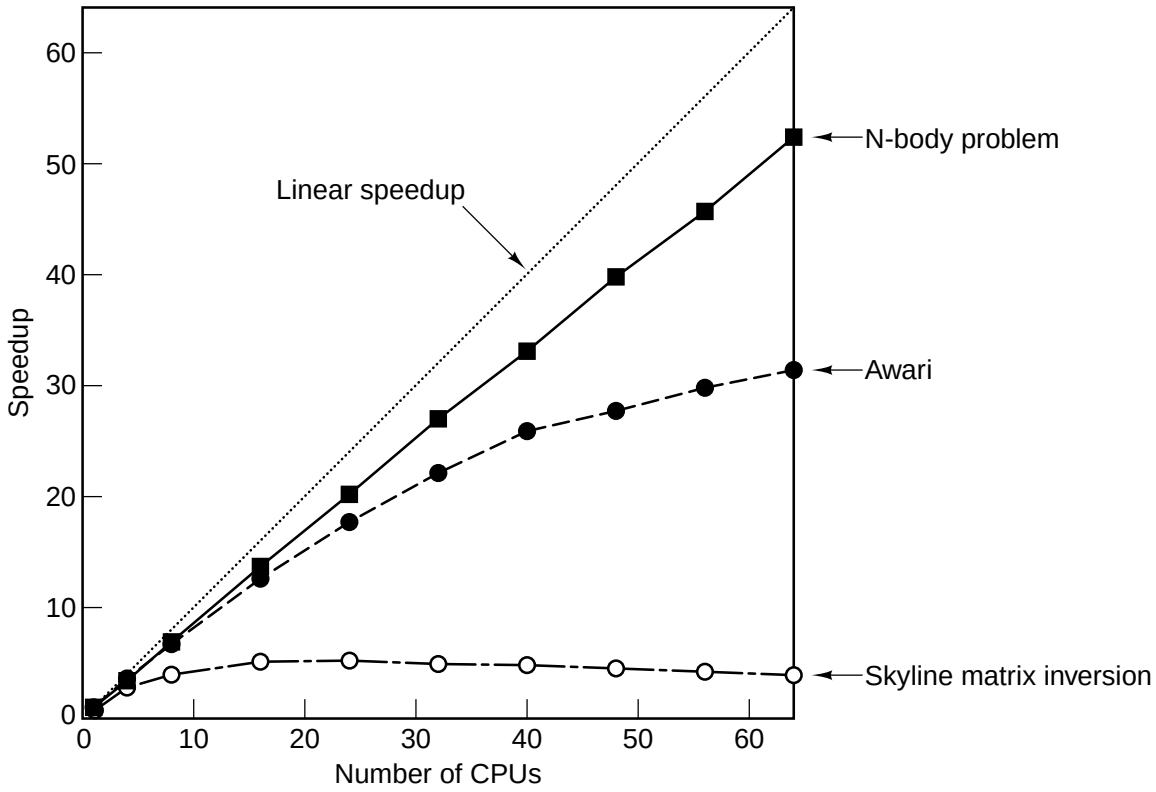


Figure 8-8. Real programs achieve less than the perfect speed-up indicated by the dotted line.

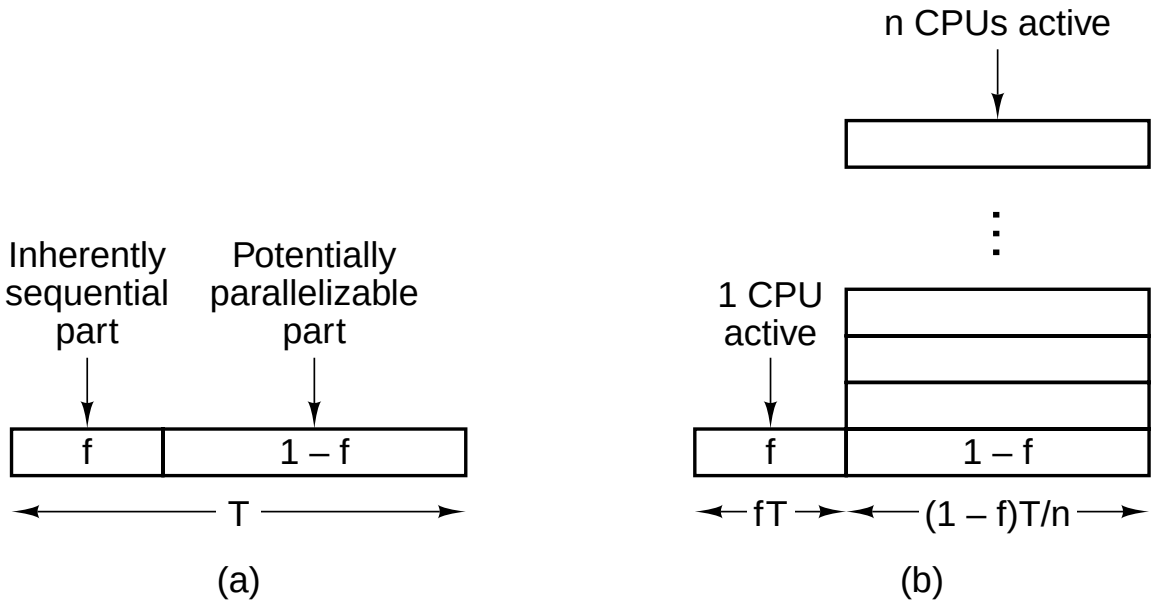


Figure 8-9. (a) A program has a sequential part and a parallelizable part. (b) Effect of running part of the program in parallel.

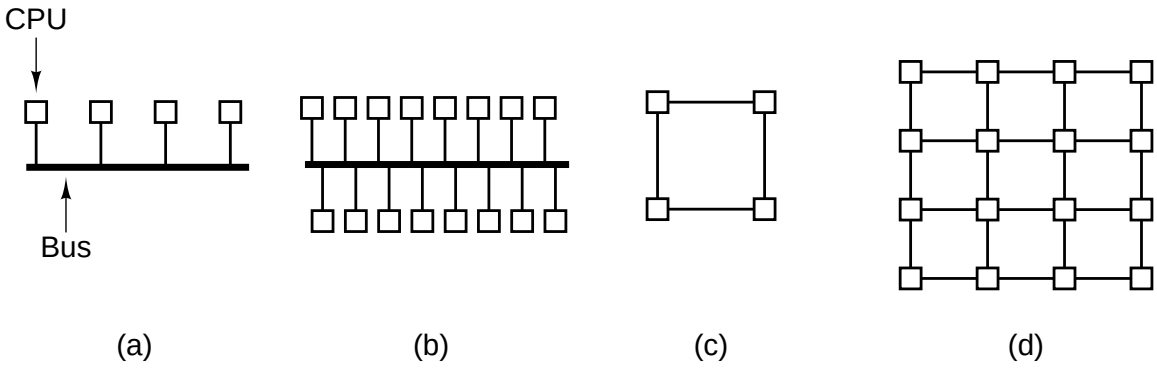


Figure 8-10. (a) A 4-CPU bus-based system. (b) A 16-CPU bus-based system. (c) A 4-CPU grid-based system. (d) A 16-CPU grid-based system.

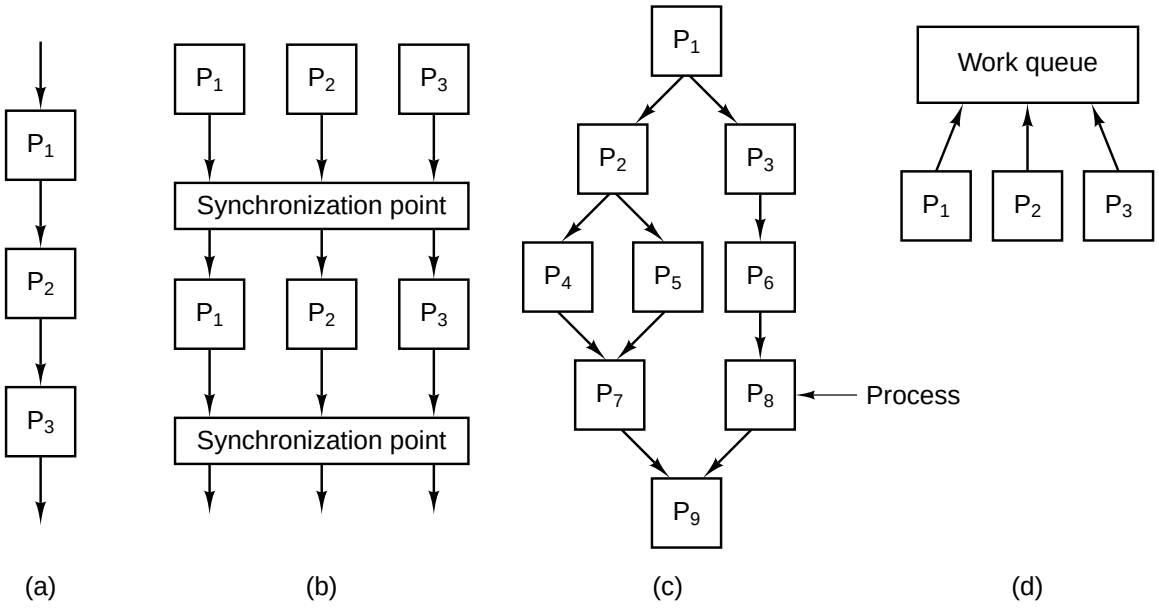


Figure 8-11. Computational paradigms. (a) Pipeline. (b) Phased computation. (c) Divide and conquer. (d) Replicated worker.

Physical (hardware)	Logical (software)	Examples
Multiprocessor	Shared variables	Image processing as in Fig. 8-1
Multiprocessor	Message passing	Message passing simulated with buffers in memory
Multicomputer	Shared variables	DSM, Linda, Orca, etc. on an SP/2 or a PC network
Multicomputer	Message passing	PVM or MPI on an SP/2 or a network of PCs

Figure 8-12. Combinations of physical and logical sharing.

Instruction streams	Data streams	Name	Examples
1	1	SISD	Classical Von Neumann machine
1	Multiple	SIMD	Vector supercomputer, array processor
Multiple	1	MISD	Arguably none
Multiple	Multiple	MIMD	Multiprocessor, multicomputer

Figure 8-13. Flynn's taxonomy of parallel computers.

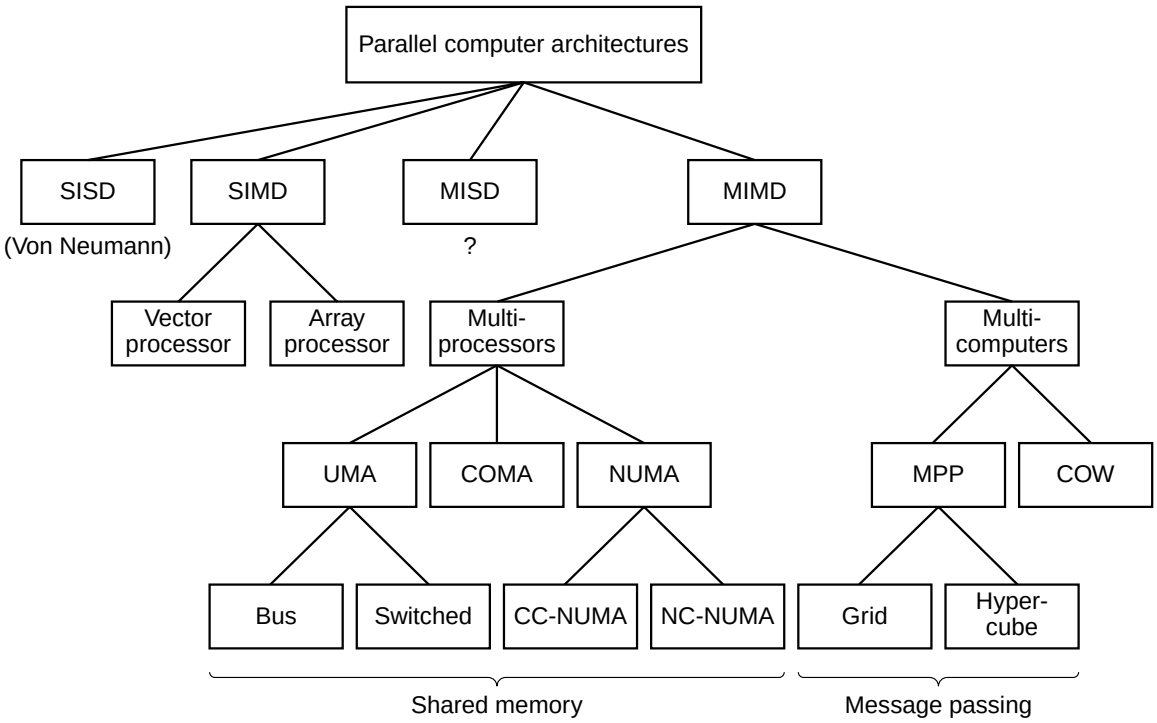


Figure 8-14. A taxonomy of parallel computers.

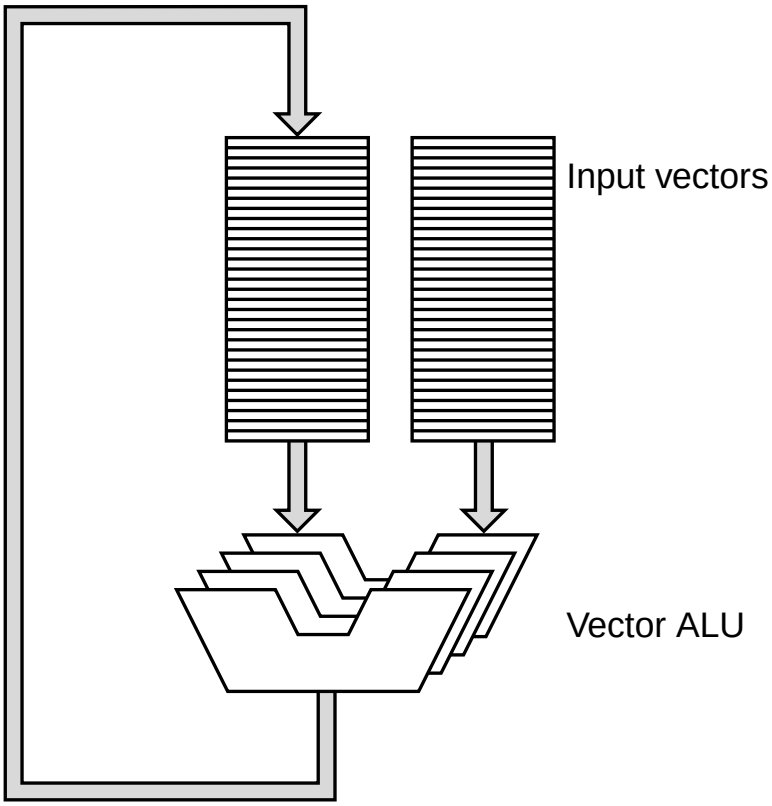


Figure 8-15. A vector ALU.

Operation	Examples
$A_i = f_1(B_i)$	f_1 = cosine, square root
Scalar = $f_2(A)$	f_2 = sum, minimum
$A_i = f_3(B_i, C_i)$	f_3 = add, subtract
$A_i = f_4(\text{scalar}, B_i)$	f_4 = multiply B_i by a constant

Figure 8-16. Various combinations of vector and scalar operations.

Step	Name	Values
1	Fetch operands	$1.082 \times 10^{12} - 9.212 \times 10^{11}$
2	Adjust exponent	$1.082 \times 10^{12} - 0.9212 \times 10^{12}$
3	Execute subtraction	0.1608×10^{12}
4	Normalize result	1.608×10^{11}

Figure 8-17. Steps in a floating-point subtraction.

	Cycle						
Step	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Fetch operands	B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	B_5, C_5	B_6, C_6	B_7, C_7
Adjust exponent		B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	B_5, C_5	B_6, C_6
Execute operation			$B_1 + C_1$	$B_2 + C_2$	$B_3 + C_3$	$B_4 + C_4$	$B_5 + C_5$
Normalize result				$B_1 + C_1$	$B_2 + C_2$	$B_3 + C_3$	$B_4 + C_4$

Figure 8-18. A pipelined floating-point adder.

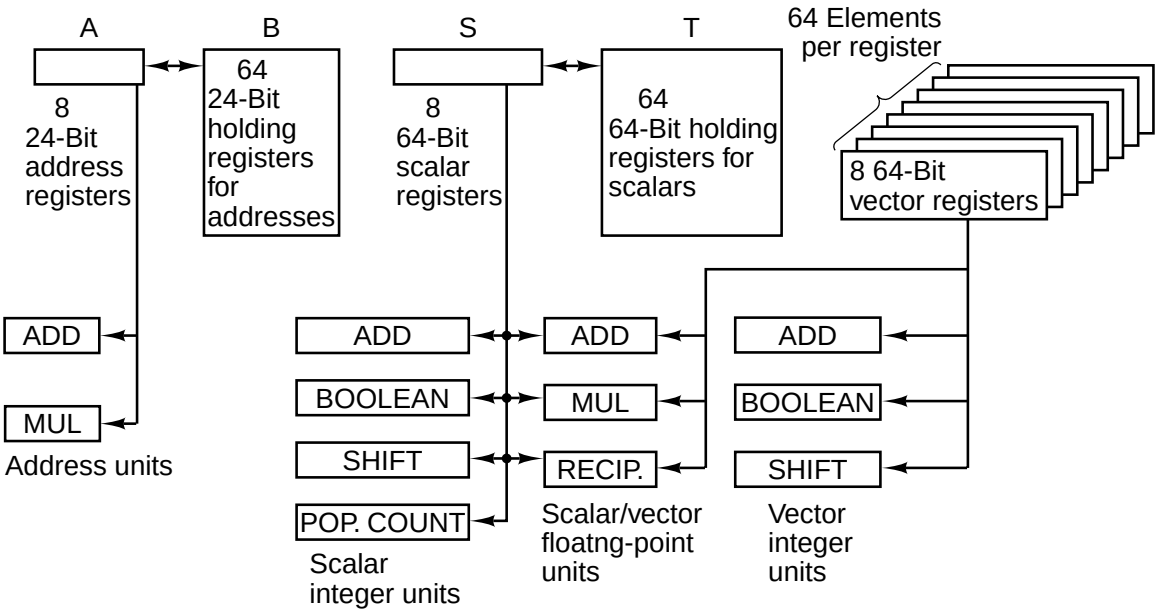


Figure 8-19. Registers and functional units of the Cray-1

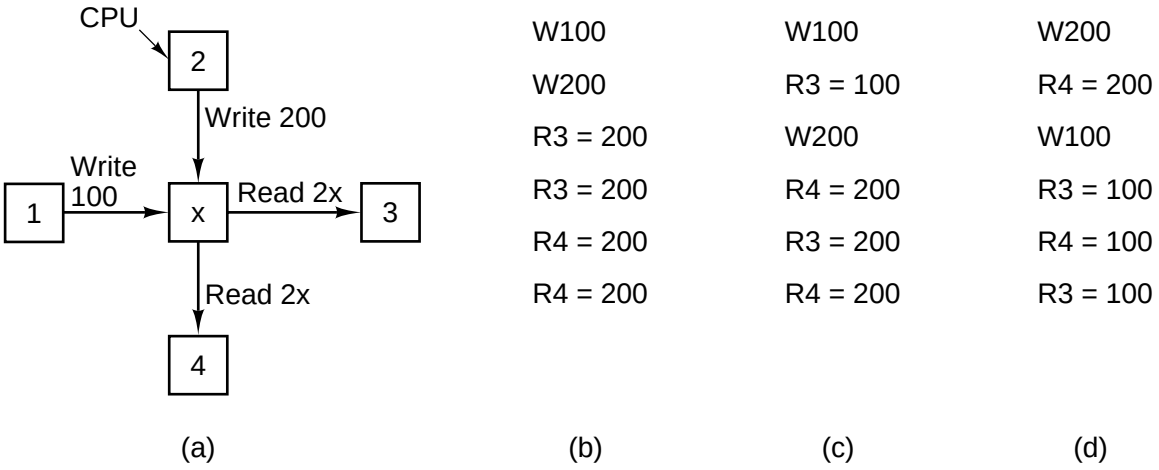


Figure 8-20. (a) Two CPUs writing and two CPUs reading a common memory word. (b) - (d) Three possible ways the two writes and four reads might be interleaved in time.

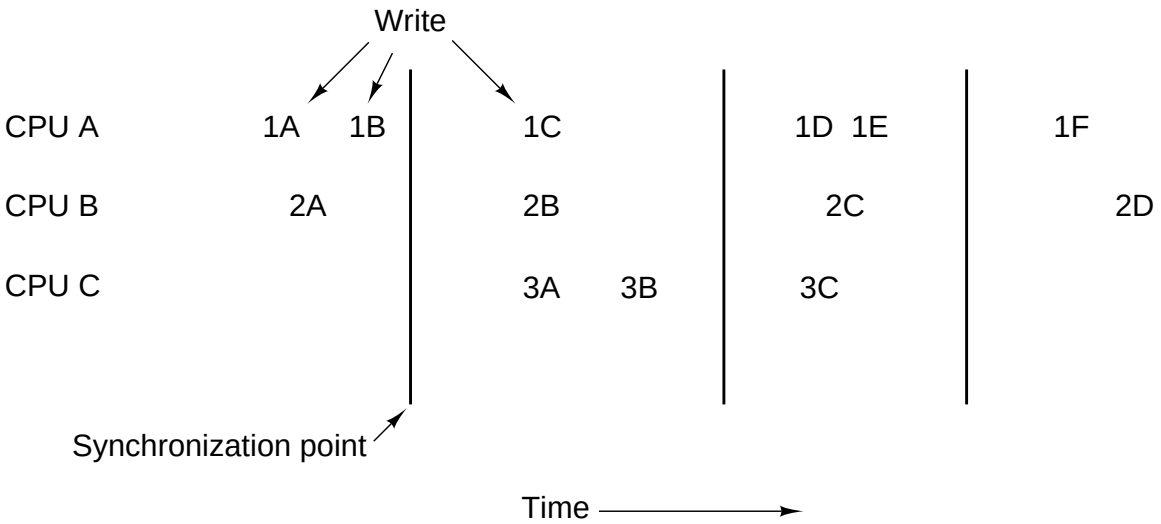


Figure 8-21. Weakly consistent memory uses synchronization operations to divide time into sequential epochs.

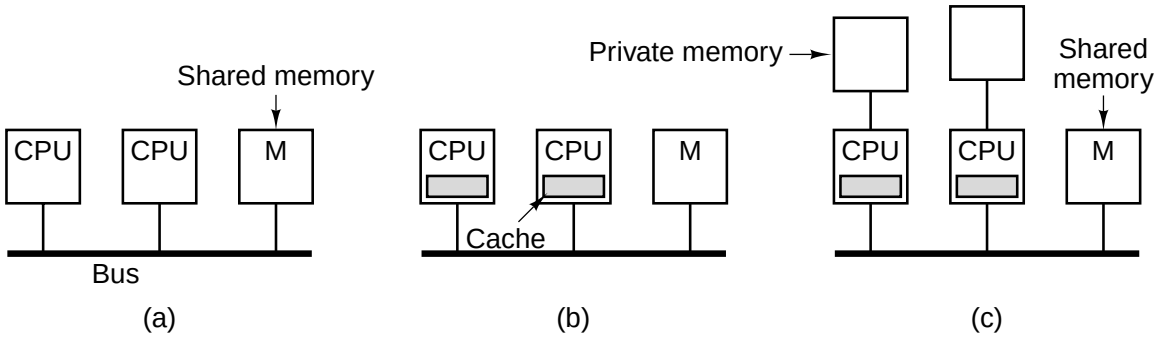


Figure 8-22. Three bus-based multiprocessors. (a) Without caching. (b) With caching. (c) With caching and private memories.

Action	Local request	Remote request
Read miss	Fetch data from memory	
Read hit	Use data from local cache	
Write miss	Update data in memory	
Write hit	Update cache and memory	Invalidate cache entry

Figure 8-23. The write through cache coherence protocol. The empty boxes indicate that no action is taken.

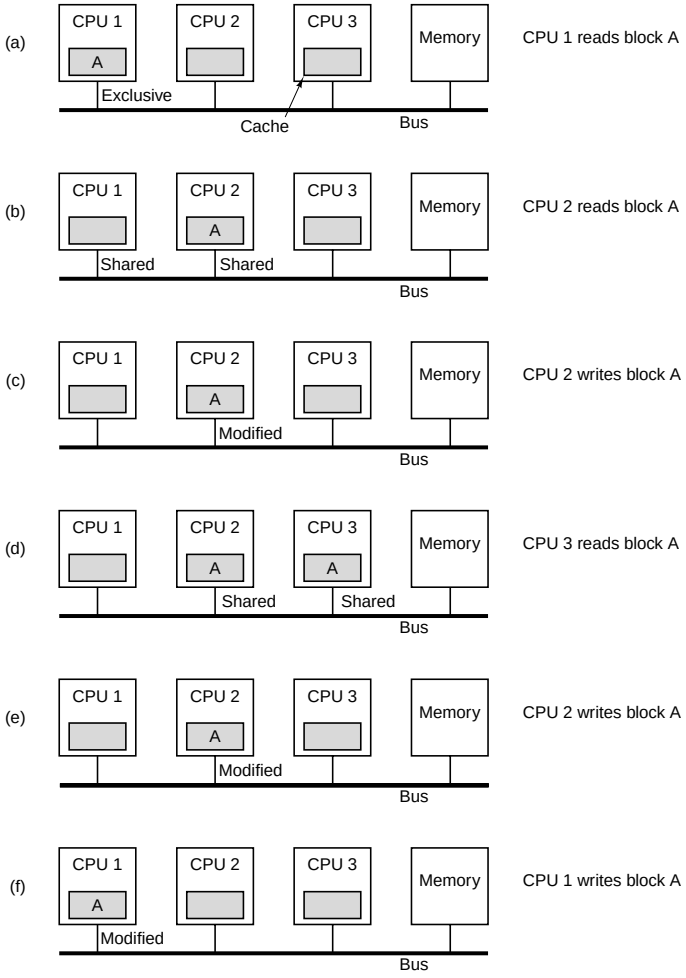


Figure 8-24. The MESI cache coherence protocol.

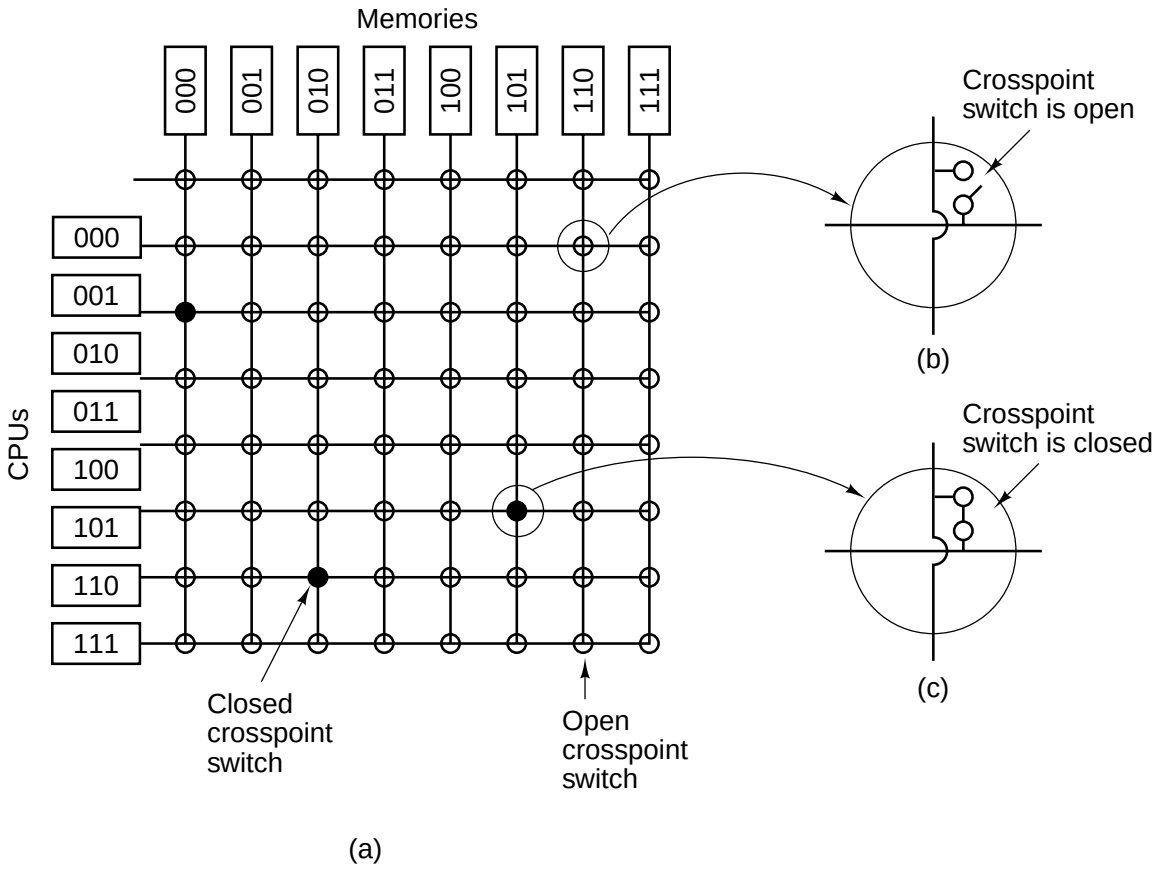


Figure 8-25. (a) An 8×8 crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.

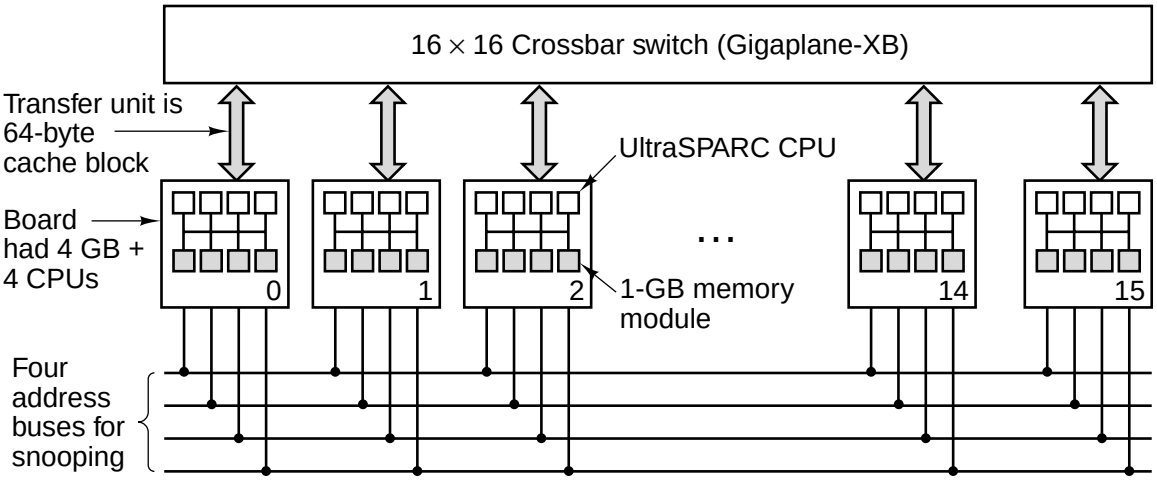


Figure 8-26. The Sun Enterprise 10000 symmetric multiprocessor.

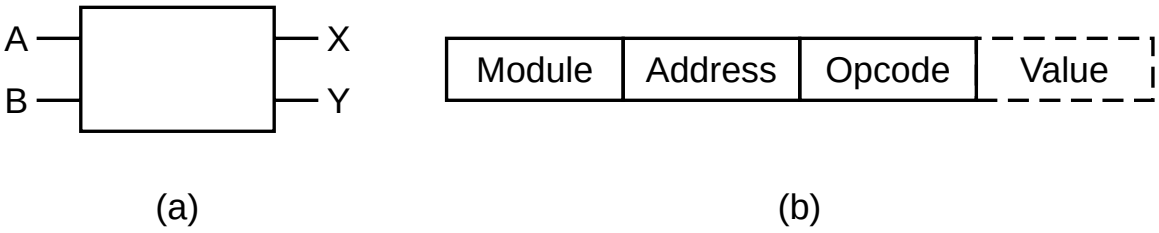


Figure 8-27. (a) A 2×2 switch. (b) A message format.

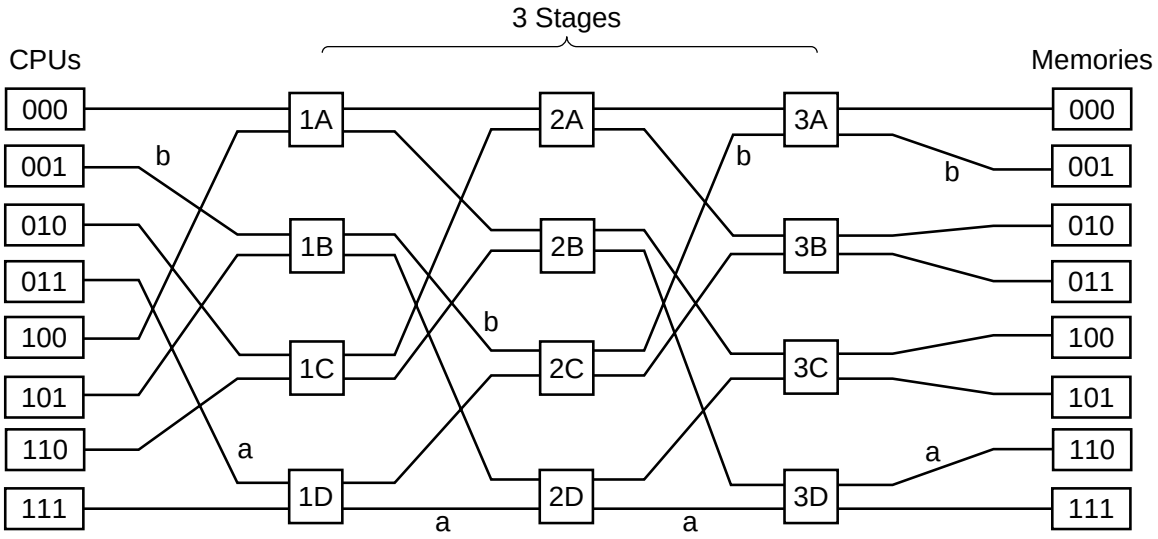


Figure 8-28. An omega switching network.

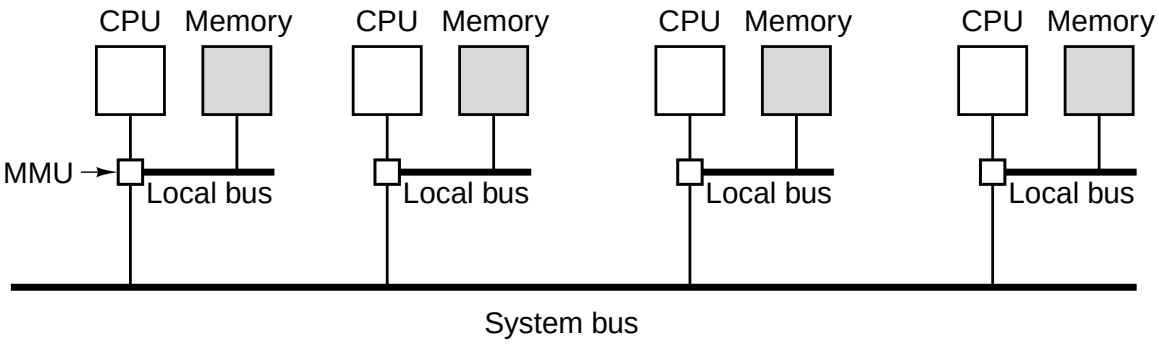


Figure 8-29. A NUMA machine based on two levels of buses. The Cm* was the first multiprocessor to use this design.

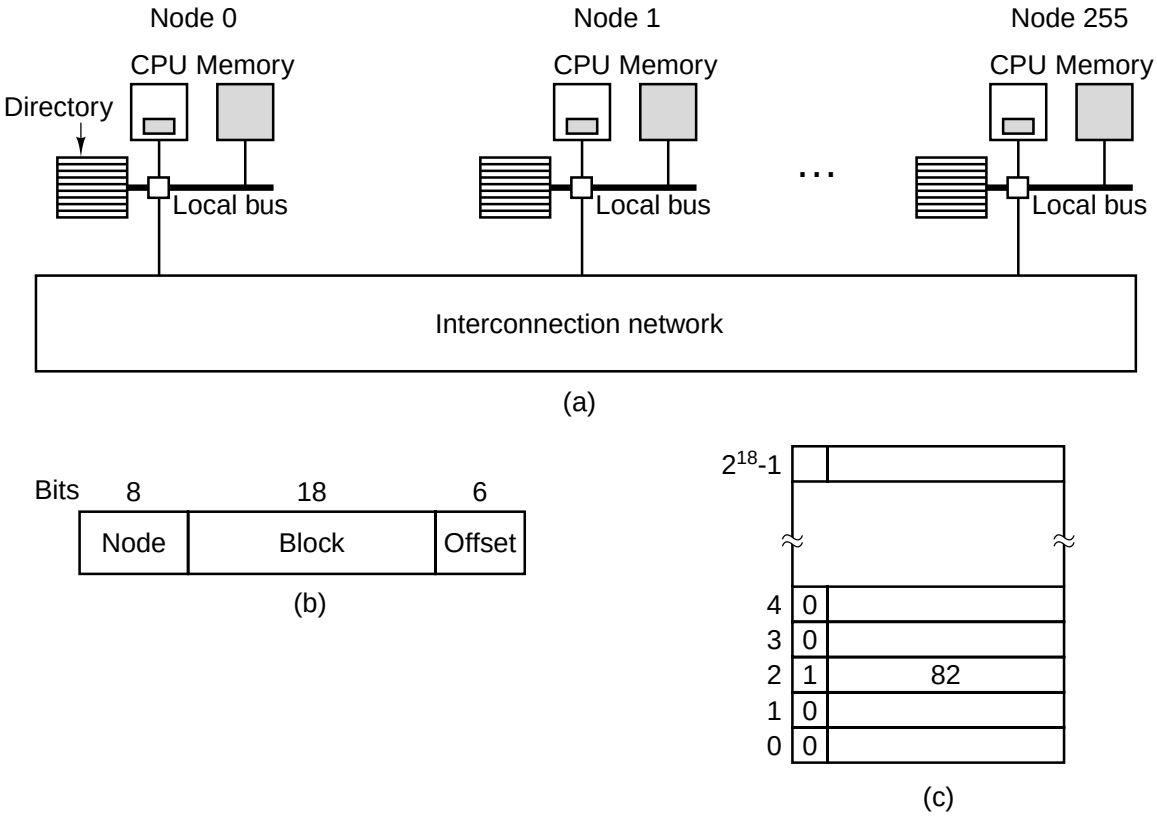


Figure 8-30. (a) A 256-node directory-based multiprocessor. (b) Division of a 32-bit memory address into fields. (c) The directory at node 36.

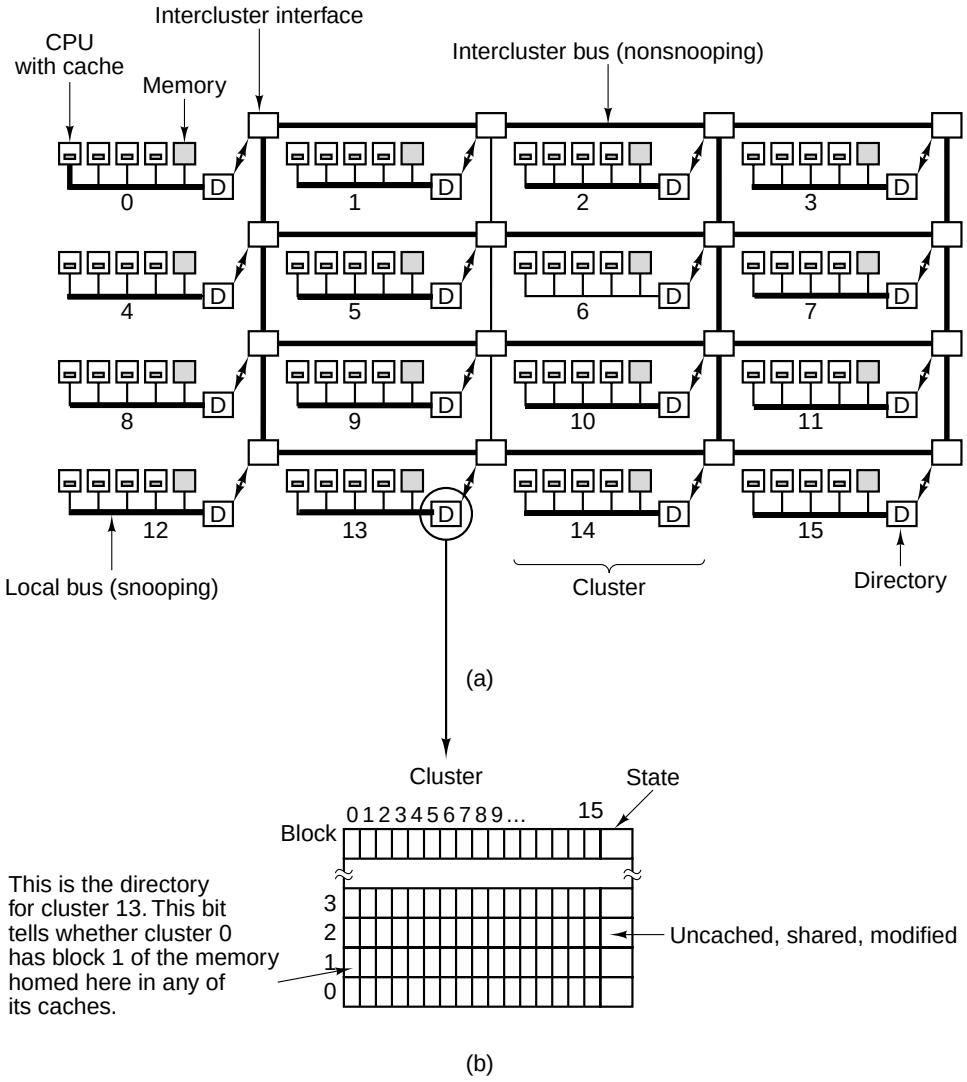


Figure 8-31. (a) The DASH architecture. (b) A DASH directory.

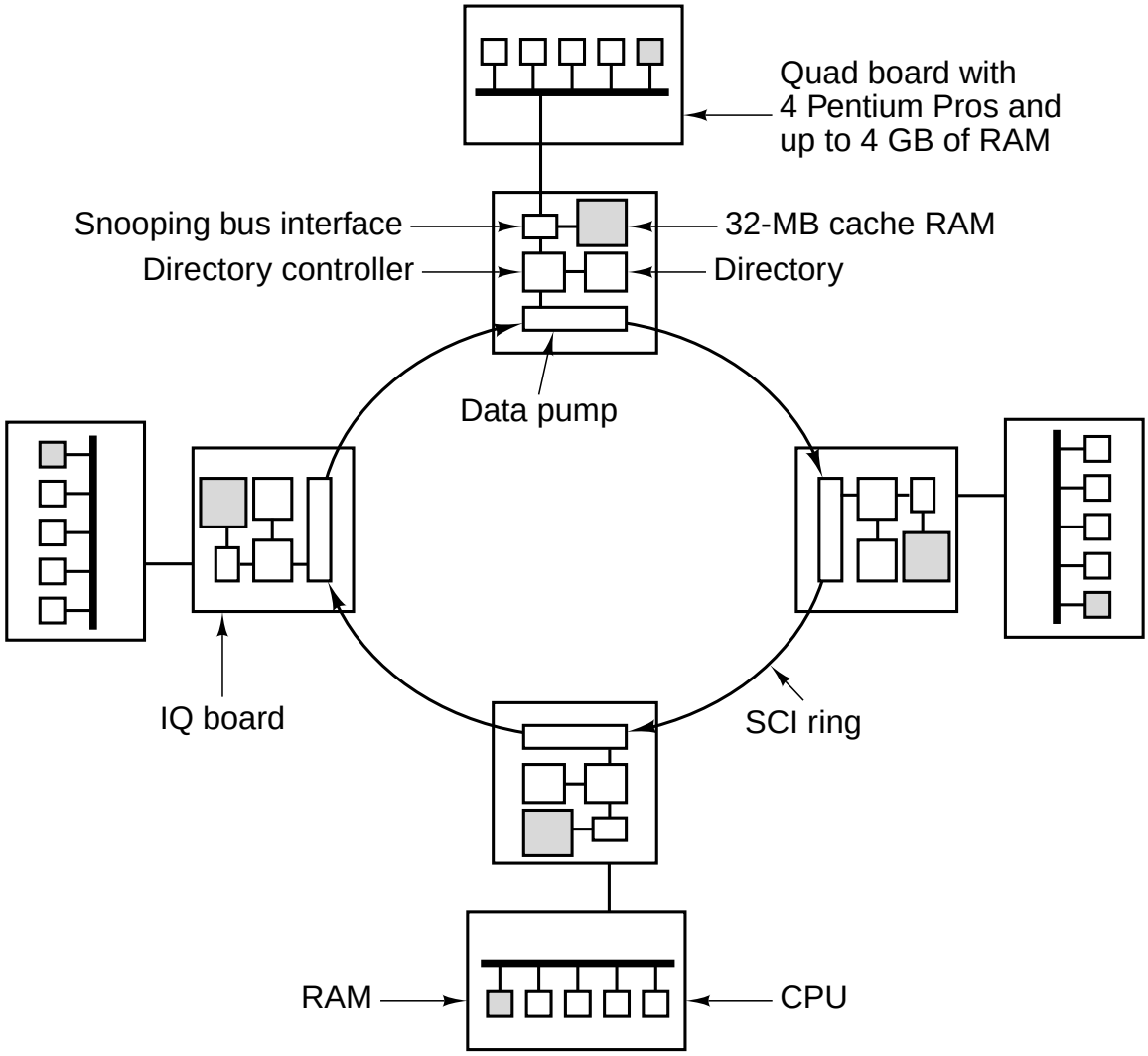


Figure 8-32. The NUMA-Q multiprocessor.

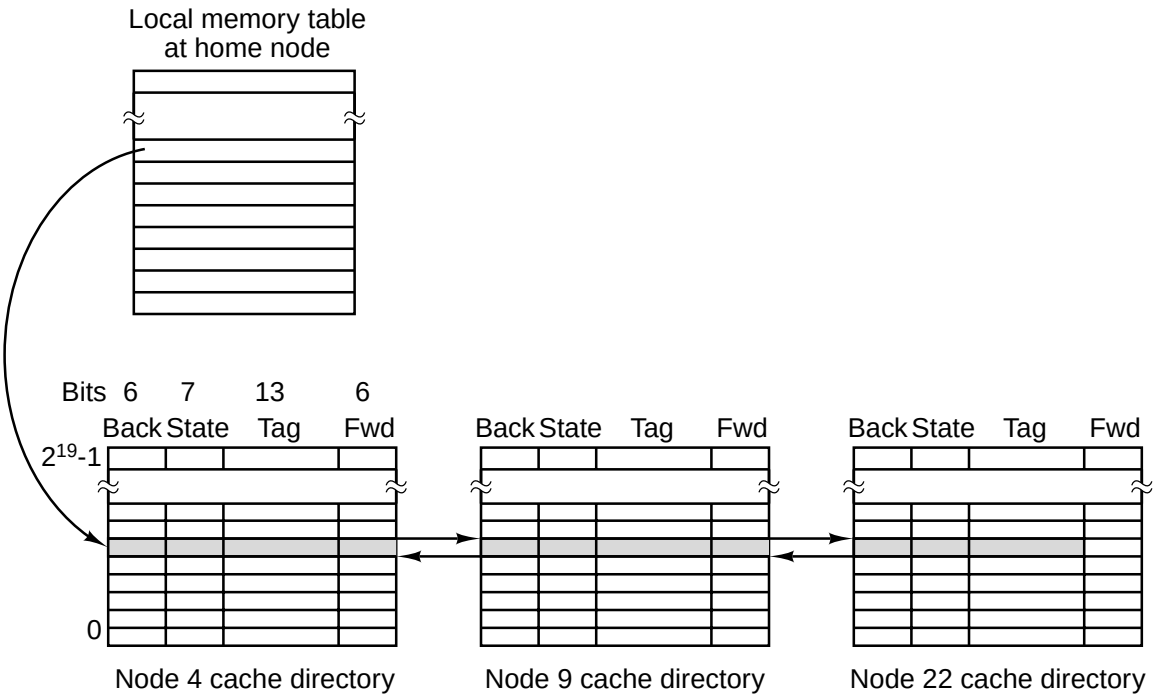


Figure 8-33. SCI chains all the holders of a given cache line together in a doubly-linked list. In this example, a line is shown cached at three nodes.

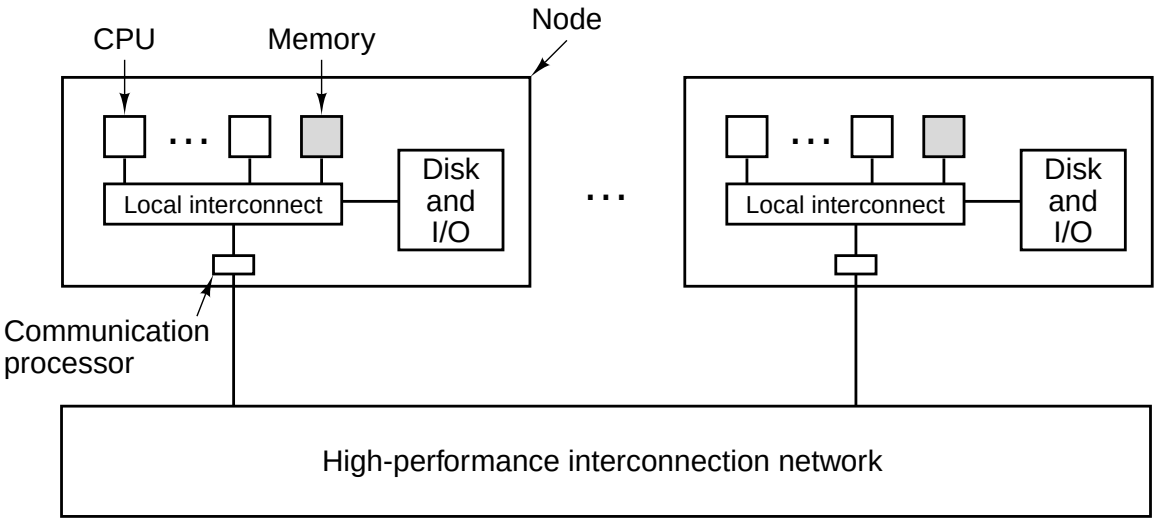


Figure 8-34. A generic multicomputer.

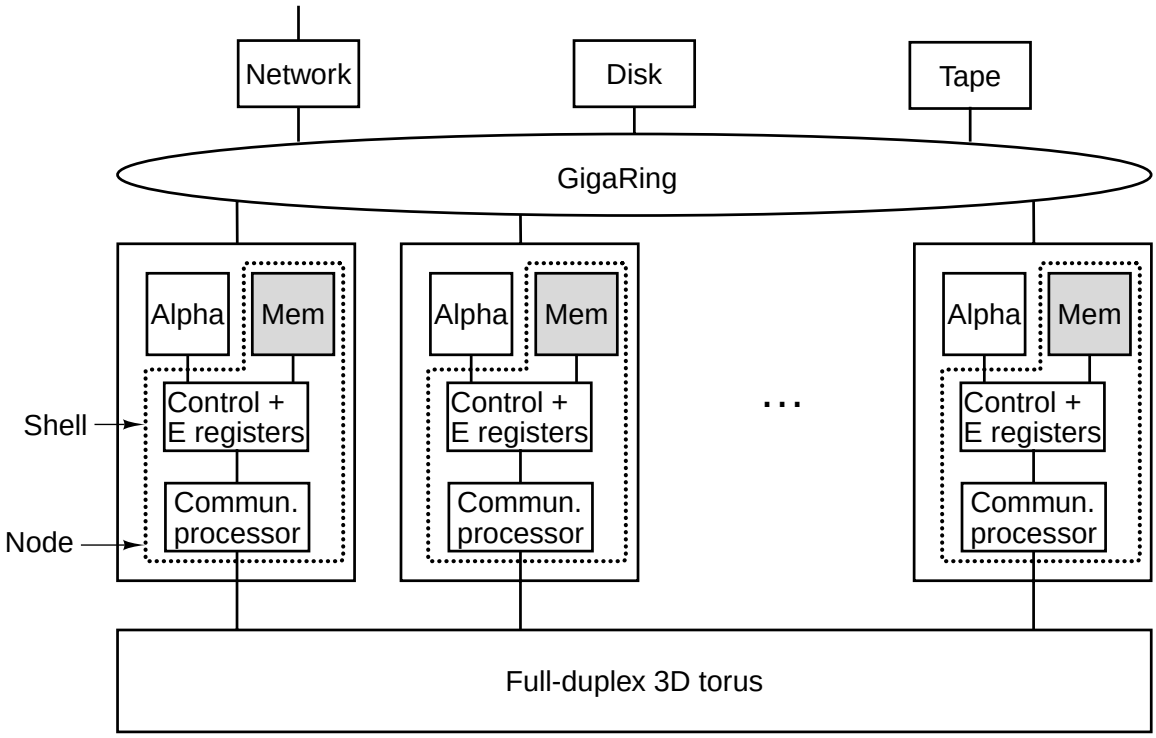


Figure 8-35. The Cray Research T3E.

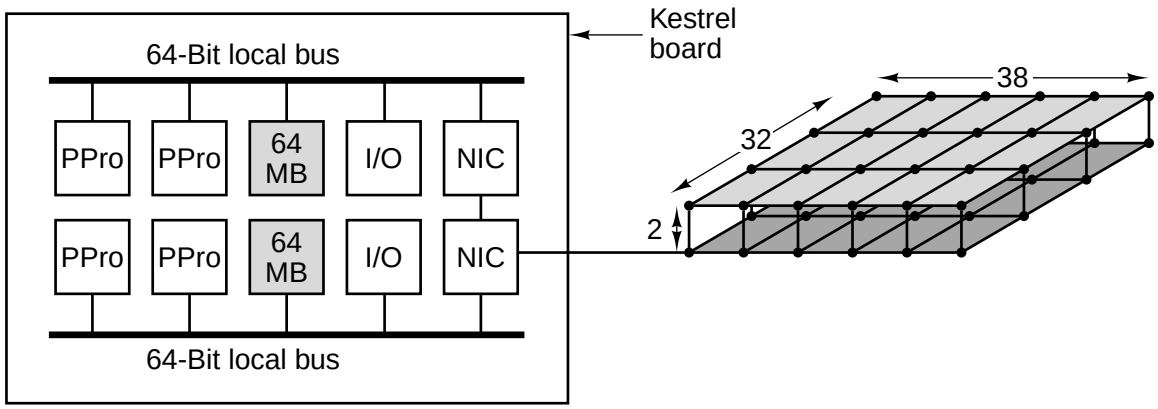


Figure 8-36. The Intel/Sandia Option Red system. (a) The kestrel board. (b) The interconnection network.

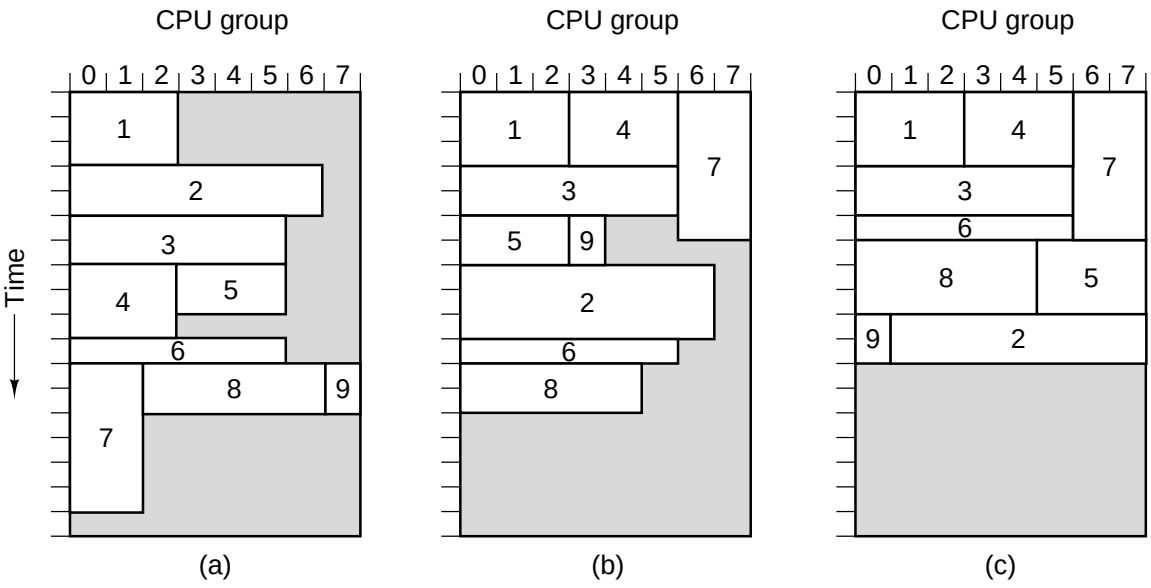


Figure 8-37. Scheduling a COW. (a) FIFO. (b) Without head-of-line blocking. (c) Tiling. The shaded areas indicate idle CPUs.

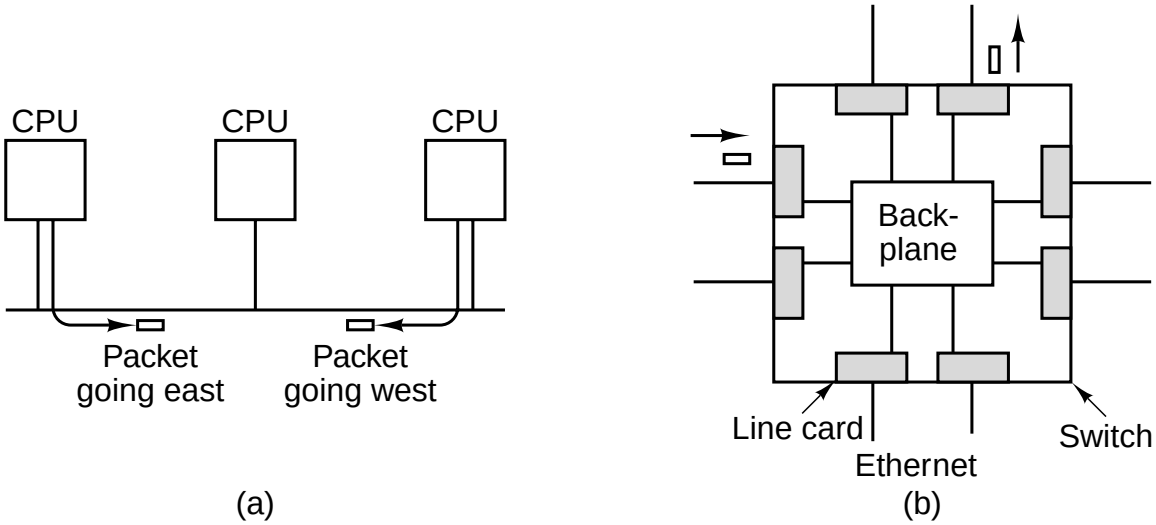


Figure 8-38. (a) Three computers on an Ethernet. (b) An Ethernet switch.

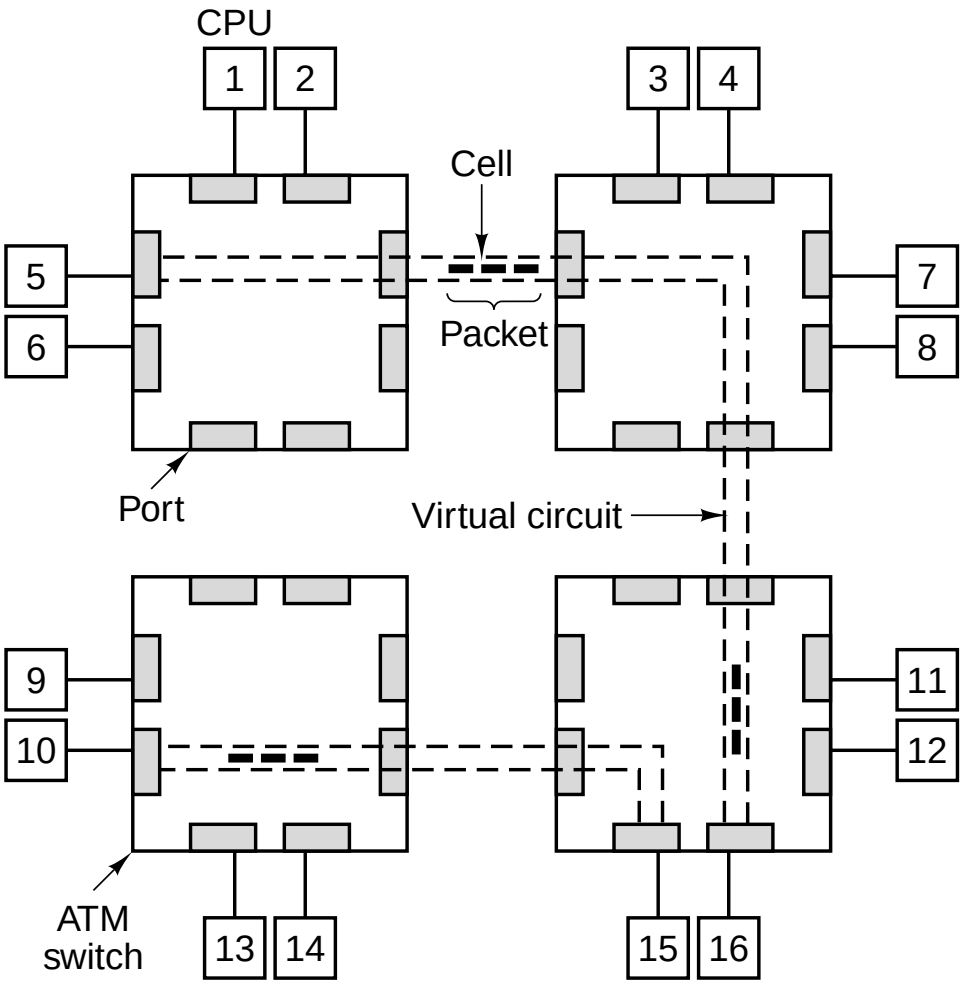


Figure 8-39. Sixteen CPUs connected by four ATM switches. Two virtual circuits are shown.

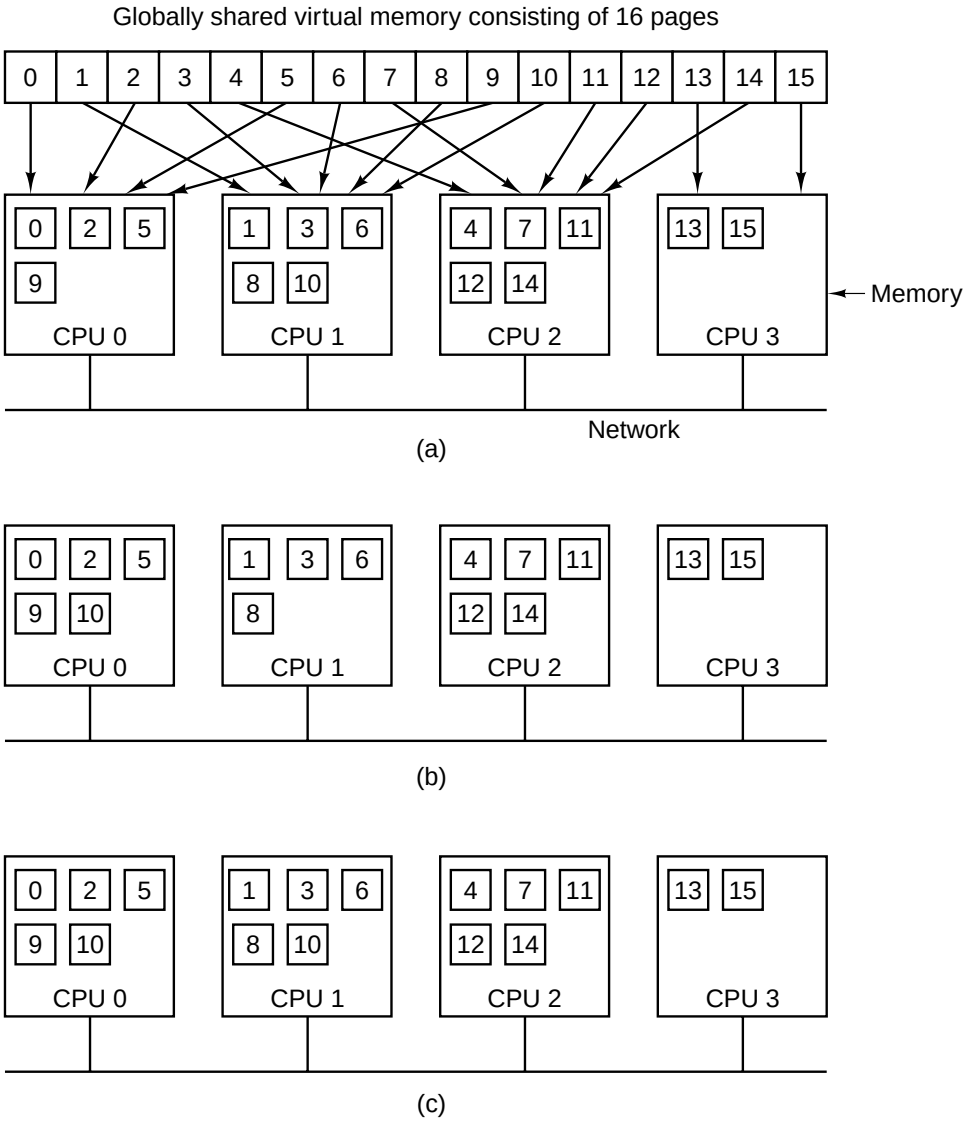


Figure 8-40. A virtual address space consisting of 16 pages spread over four nodes of a multicomputer. (a) The initial situation. (b) After CPU 0 references page 10. (c) After CPU 1 references page 10, here assumed to be a read-only page.

(″abc″, 2, 5)
(″matrix-1″, 1, 6, 3.14)
(″family″, ″is sister″, Carolyn, Elinor)

Figure 8-41. Three Linda tuples.

```
Object implementation stack;
   top:integer;                    # storage for the stack
   stack: array [integer 0..N-1] of integer;

   operation push(item: integer);  function returning nothing
   begin
      stack[top] := item;          push item onto the stack
      top := top + 1;              # increment the stack pointer
   end;

   operation pop( ): integer; # function returning an integer
   begin
      guard top > 0 do             # suspend if the stack is empty
         top := top - 1;           # decrement the stack pointer
         return stack[top];        # return the top item
      od;
   end;

begin
   top := 0;                       # initialization
end;
```

Figure 8-42. A simplified ORCA stack object, with internal data and two operations.

8.1 DESIGN ISSUES FOR PARALLEL COMPUTERS

When approaching a new parallel computer system, three fundamental questions to ask are

1. What are the nature, size, and number of the processing elements?
2. What are the nature, size, and number of the memory modules?
3. How are the processing and memory elements interconnected?

Let us examine each of these points briefly in turn. The processing elements can range from minimal ALUs through complete CPUs, with sizes ranging from a small portion of a chip to a cubic meter of electronics per element. As one might expect, when the processing element is a fraction of a chip, it is possible to equip a computer with vast numbers, perhaps even a million of such elements. When the processing element is a complete computer, with its own memory and I/O equipment, the sizes are smaller, naturally, although systems with almost 10,000 CPUs have been installed. More and more, parallel computers are being constructed from commercially available parts, especially CPUs. The capabilities and limitations of these parts often heavily influence the design.

Memory systems are often split up into modules that operate independently of one another, in parallel to allow access by many CPUs at the same time. These modules can be small (kilobytes) or large (megabytes) and closely integrated with the CPUs or located on a different circuit board. Since large dynamic memories (DRAMs) are usually much slower than CPUs, elaborate caching schemes are frequently used to speed up memory access. Often two, three, and even four levels of caching are used.

Although there is some variation among CPU and memory designs, the area in which parallel systems differ most is how the pieces are put together. The interconnection schemes can be divided into two rough categories: static and dynamic. The static ones simply wire up all the components in a fixed way, such as a star, ring, or grid. In dynamic interconnection schemes, all the pieces are hooked up to a switching network that can dynamically route messages between components. Each has its own strengths and weaknesses, as we shall soon see.

Viewing parallel computers as a collection of chips that are wired up in one manner or another is essentially a bottom-up view of the world. In a top-down approach, one would ask: ‘‘What is it that is to be run in parallel?’’ Here again, we have a spectrum of possibilities. Some parallel computers are designed to run multiple independent jobs simultaneously. These jobs have nothing to do with one another and do not communicate. A typical example is a computer with 8 to 64 CPUs intended as a big UNIX timesharing system for handling thousands of remote users. Transaction processing systems used by banks (e.g., automated teller machines), airlines (e.g., reservation systems) and large Web servers also fall into this category, as do independent simulation runs using different sets of

parameters.

A different point on this spectrum is the parallel computer used for running a single job consisting of many parallel processes. As an example, consider a chess program that analyzes a given board by generating a list of legal moves that can be made from it, and then forking off parallel processes to (recursively) analyze each new board in parallel. The point of the parallelism here is not to accommodate more users, but to gain speedup on a single problem.

Continuing along this spectrum, we come to machines in which the parallelism comes from a high degree of pipelining or many ALUs operating on the same instruction stream at the same time. Numeric supercomputers with special hardware for vector processing fall in this category. Here we not only have one main problem being solved, but all parts of the computer are working very closely on almost the same aspect of the problem together (e.g., different elements of the same two vectors are being added in parallel).

Although it is hard to pin down exactly, these three examples differ in what is sometimes called **grain size**. In the multi-CPU timesharing systems, the unit of parallelism is large: a complete user program. Running large pieces of software in parallel with little or no communication between the pieces is called **coarse-grained parallelism**. The opposite extreme, such as found in vector processing, is called **fine-grained parallelism**.

Grain size refers to the algorithms and software, but it has a direct analog in the hardware. Systems with a small number of large, independent CPUs that have low-speed connections between the CPUs are called **loosely coupled**. Their opposite number are the **tightly-coupled** systems, in which the components are generally smaller, closer together and interact with each other frequently over high-bandwidth communication networks. In most cases, problems with coarse-grained parallelism work best on loosely-coupled systems; Similarly, problems with fine-grained parallelism work best on tightly-coupled systems. However, so much variety exists in algorithms, software, and hardware, that this is a general guide at best. The wide range in problem grain size and possibilities for system coupling have led to the wide variety of architectures that we will study in this chapter.

In the following sections we will examine some of the design issues for parallel computers, starting with communication models and interconnection networks. Then we will look at performance and software issues. Finally, we will conclude with a taxonomy of parallel computer architectures that will guide the organization of the rest of this chapter.

8.1.1 Communication Models

In any parallel computer system, CPUs working on different parts of the same job must communicate with one another to exchange information. Precisely how they should do this is the subject of much debate in the architectural community.

Two distinct designs, multiprocessors and multicomputers, have been proposed and implemented. In this section we will examine each of them.

Multiprocessors

In the first design, all CPUs share a common physical memory, as illustrated in Fig. 8-1(a). A system based on shared memory, like this one, is called a **multiprocessor** or sometimes just a **shared memory system**.

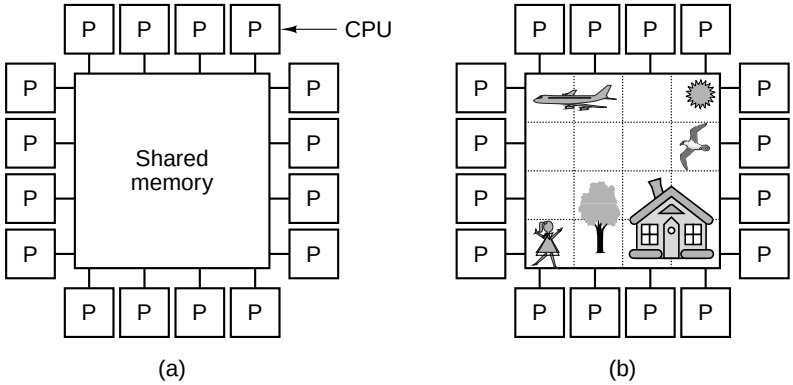


Figure 8-1. (a) A multiprocessor with 16 CPUs sharing a common memory. (b) An image partitioned into 16 sections, each being analyzed by a different CPU.

The multiprocessor model extends into software. All processes working together on a multiprocessor can share a single virtual address space mapped onto the common memory. Any process can read or write a word of memory by just executing a LOAD or STORE instruction. Nothing else is needed. Two processes can communicate by simply having one of them write data to memory and having the other one read them back.

The ability for two (or more) processes to communicate by just reading and writing memory is the reason multiprocessors are popular. It is an easy model for programmers to understand and is applicable to a wide range of problems. Consider, for example, a program that inspects a bit-map image and lists all the objects in it. One copy of the image is kept in memory, as shown in Fig. 8-1(b). Each of the 16 CPUs runs a single process, which has been assigned one of the 16 sections to analyze. Nevertheless, each process has access to the entire image, which is essential, since some objects occupy multiple sections. If a process discovers that one of its objects extends over a section boundary, it just follows the object into the next section by reading the words of that section. In this example, some objects will be discovered by multiple processes, so some coordination is needed at the end to determine how many houses, trees, and airplanes there are.

Many computer vendors sell multiprocessors. Some examples are the Sun

Enterprise 10000, Sequent NUMA-Q, SGI Origin 2000, and HP/Convex Exemplar.

Multicomputers

The second design for a parallel architecture is one in which each CPU has its own private memory, accessible only to itself and not to any other CPU. Such a design is called a **multicomputer**, or sometimes a **distributed memory system**, and is illustrated in Fig. 8-2(a). Multicomputers are frequently (but not always) loosely coupled. The key aspect of a multicomputer that distinguishes it from a multiprocessor is that each CPU in a multicomputer has its own private, local memory that it can access by just executing LOAD and STORE instructions, but which no other CPU can access using LOAD and STORE instructions. Thus multiprocessors have a single physical address space shared by all the CPUs whereas multicomputers have one physical address space per CPU.

Since the CPUs on a multicomputer cannot communicate by just reading and writing the common memory, they need a different communication mechanism. What they do is pass messages back and forth using the interconnection network. Examples of multicomputers include the IBM SP/2, Intel/Sandia Option Red, and the Wisconsin COW.

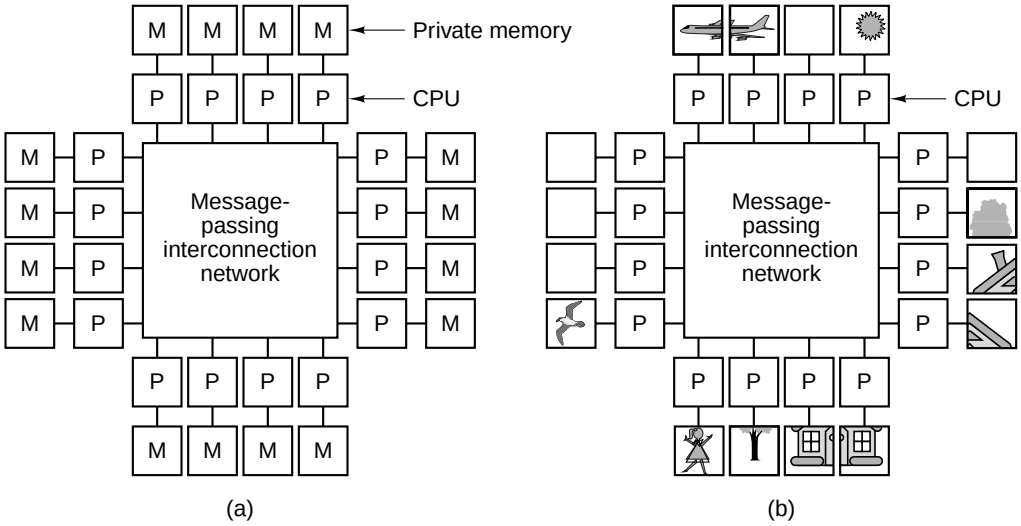


Figure 8-2. (a) A multicomputer with 16 CPUs, each with each own private memory. (b) The bit-map image of Fig. 8-1 split up among the 16 memories.

The absence of hardware shared memory on a multicomputer has important implications for the software structure. Having a single virtual address space with all processes being able to read and write all of memory by just executing LOAD

and STORE instructions is impossible on a multicomputer. For example, if CPU 0 (the one in the upper left-hand corner) of Fig. 8-1(b) discovers that part of its object extends into the section assigned to CPU 1, it can nevertheless just continue reading memory to access the nose of the airplane. On the other hand, if CPU 0 in Fig. 8-2(b) makes the same discovery, it cannot just read CPU 1's memory. It has to do something quite different to get the data it needs.

In particular, it has to discover (somehow) which CPU has the data it needs and send that CPU a message requesting a copy of the data. Typically it will then block (i.e., wait) until the request is answered. When the message arrives at CPU 1, software there has to analyze it and send back the needed data. When the reply message gets back to CPU 0, the software is unblocked and can continue executing.

On a multicomputer, communication between processes often uses software primitives such as send and receive. This gives the software a different, and far more complicated structure, than on a multiprocessor. It also means that correctly dividing up the data and placing them in the optimal locations is a major issue on a multicomputer. It is less of an issue on a multiprocessor since placement does not affect correctness or programmability although it may affect performance. In short, programming a multicomputer is much more difficult than programming a multiprocessor.

Under these conditions, why would anyone build multicomputers, when multiprocessors are easier to program? The answer is simple: large multicomputers are much simpler and cheaper to build than multiprocessors with the same number of CPUs. Implementing a memory shared by even a few hundred CPUs is a substantial undertaking, whereas building a multicomputer with 10,000 CPUs or more is straightforward.

Thus we have a dilemma: multiprocessors are hard to build but easy to program whereas multicomputers are easy to build but hard to program. This observation has led to a great deal of effort to construct hybrid systems that are relatively easy to build and relatively easy to program. This work has led to the realization that shared memory can be implemented in various ways, each with its own set of advantages and disadvantages. In fact, much of the research in parallel architectures these days relates to the convergence of multiprocessor and multicomputer architectures into hybrid forms that combine the strengths of each. The holy grail here is to find designs that are **scalable**, that is, continue to perform well as more and more CPUs are added.

One approach to building hybrid systems is based on the fact that modern computer systems are not monolithic but are constructed as a series of layers—the theme of this book. This insight opens the possibility of implementing the shared memory at any one of several layers, as shown in Fig. 8-3. In Fig. 8-3(a) we see the shared memory being implemented by the hardware as a true multiprocessor. In this design, there is a single copy of the operating system with a single set of tables, in particular, the memory allocation table. When a process needs more

memory, it traps to the operating system, which then looks in its table for a free page and maps the page into the caller's address space. As far as the operating system is concerned, there is a single memory and it keeps track of which process owns which page in software. There are many ways to implement hardware shared memory, as we will see later.

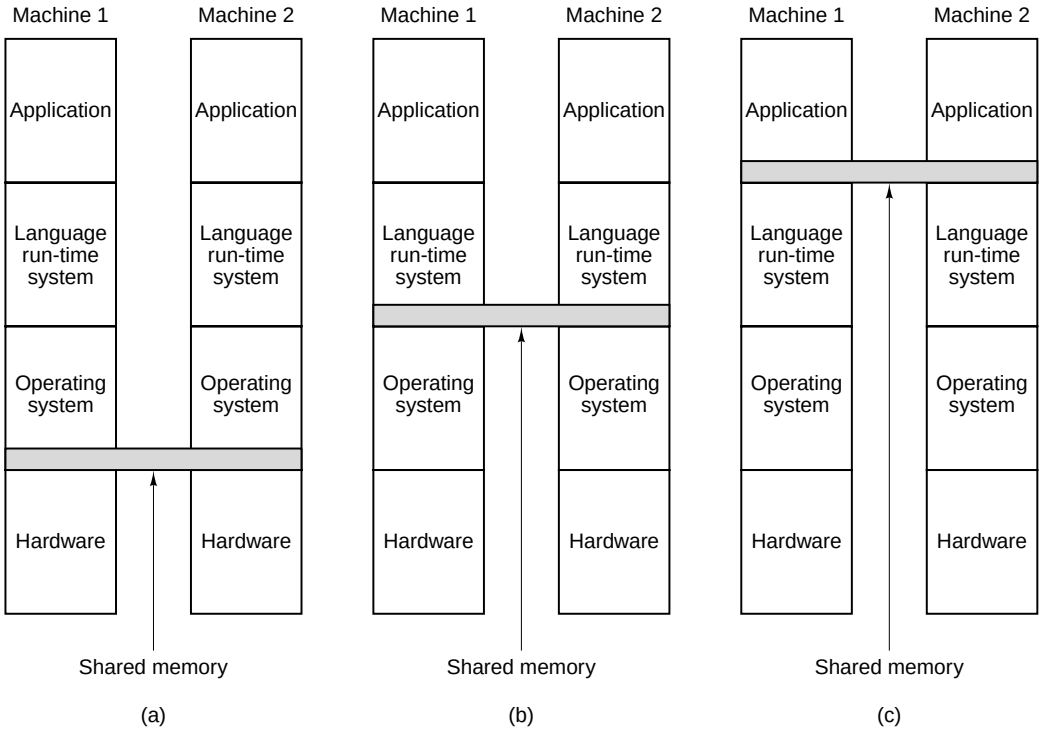


Figure 8-3. Various layers where shared memory can be implemented. (a) The hardware. (b) The operating system. (c) The language runtime system.

A second possibility is to use multicomputer hardware and have the operating system simulate shared memory by providing a single system-wide paged shared virtual address space. In this approach, called **DSM** (**Distributed Shared Memory**) (Li, 1988; and Li and Hudak, 1986, 1989), each page is located in one of the memories of Fig. 8-2(a). Each machine has its own virtual memory and its own page tables. When a CPU does a LOAD or STORE on a page it does not have, a trap to the operating system occurs. The operating system then locates the page and asks the CPU currently holding it to unmap the page and send it over the interconnection network. When it arrives, the page is mapped in and the faulting instruction is restarted. In effect, the operating system is just satisfying page faults from remote memory instead of from disk. To the user, the machine looks as if it has shared memory. We will examine DSM later in this chapter.

A third possibility is to have a user-level runtime system implement a (possibly language-specific) form of shared memory. In this approach, the programming language provides some kind of shared memory abstraction, which is then implemented by the compiler and runtime system. For example, the Linda model is based on the abstraction of a shared space of tuples (data records containing a collection of fields). Processes on any machine can input a tuple from the shared tuple space or output a tuple to the shared tuple space. Because access to the tuple space is controlled entirely in software (by the Linda runtime system), no special hardware or operating system support is needed.

Another example of a language-specific shared memory implemented by the runtime system is the Orca model of shared data objects. In Orca, processes share generic objects rather than just tuples and can execute object-specific methods on them. When a method changes the internal state of an object, it is up to the runtime system to make sure all copies of the object on all machines are simultaneously updated. Again, because objects are a strictly software concept, the implementation can be done by the runtime system help from the operating system or hardware. We will look at both Linda and Orca later in this chapter.

8.1.2 Interconnection Networks

In Fig. 8-2 we saw that multicomputers are held together by interconnection networks. Now it is time to look more closely at these interconnection networks. Interestingly enough, multiprocessors and multicomputers are surprisingly similar in this respect because multiprocessors often have multiple memory modules that must also be interconnected with one another and with the CPUs. Thus the material in this section frequently applies to both kinds of systems.

The fundamental reason why multiprocessor and multicomputer interconnection networks are similar is that at the very bottom both of them use message passing. Even on a single-CPU machine, when the processor wants to read or write a word, what it typically does is assert certain lines on the bus and wait for a reply. This action is fundamentally like message passing: the initiator sends a request and waits for a response. In large multiprocessors, communication between CPUs and remote memory almost always consists of the CPU sending an explicit message, called a **packet**, to memory requesting some data, and the memory sending back a reply packet.

Interconnection networks can consist of up to five components:

1. CPUs
2. Memory modules.
3. Interfaces.
4. Links.

5. Switches.

We have already looked at CPUs and memories in some detail in this book and will not do that further here. Basically, they are the endpoints of all communication. Interfaces are the devices that get the messages into and out of the CPUs and memories. In many designs, an interface is a chip or board that is attached to each CPU's local bus and can talk to it and to local memory, if any. Often it has a programmable processor inside of it as well as some RAM private to it alone. Usually, the interface has the ability to read and write various memories, in order to move blocks of data around.

The links are the physical channels over which the bits move. They can be electrical or optical fiber and serial (1-bit wide) or parallel (more than 1-bit wide). Each link has a maximum bandwidth, which is the number of bits per second it is capable of transferring. Links can be simplex (unidirectional), half duplex (one way at a time), or full duplex (both ways at once).

The switches are devices with multiple input ports and multiple output ports. When a packet arrives at a switch on an input port, some bits in the packet are used to select the output port to which the packet is sent. A packet might be as short as 2 or 4 bytes, but it might also be much longer (e.g., 8 KB).

A certain analogy exists between an interconnection network and the streets in a city. The streets are like links. Each has a directionality (one way or two way), a maximum data rate (speed limit), and width (number of lanes). The intersections are like switches. At each intersection, an arriving packet (pedestrian or vehicle) has a choice of which outgoing port (street) it will use, depending on where its final destination is.

When designing or analyzing an interconnection network, several areas stand out as being important. First, there is the matter of topology, that is, how the components are arranged. Second is how the switching works and how to handle contention for resources. Third is what routing algorithm is used to get messages to their destination efficiently. We will examine each of these topics briefly below.

Topology

The topology of an interconnection network describes how the links and switches are arranged, for example, as a ring or as a grid. Topological designs can be modeled as graphs, with the links as arcs and the switches as nodes, as shown in Fig. 8-4. Each node in an interconnection network (or its graph) has some number of links connected to it. Mathematicians call the number of links the **degree** of the node; engineers call it the **fanout**. In general, the greater the fanout, the more routing choices there are and the greater the fault tolerance, that is, the ability to continue functioning even if a link fails by routing around it. If every node has k arcs and the wiring is done right, it is possible to design the network so that it remains fully connected even if $k - 1$ links fail.

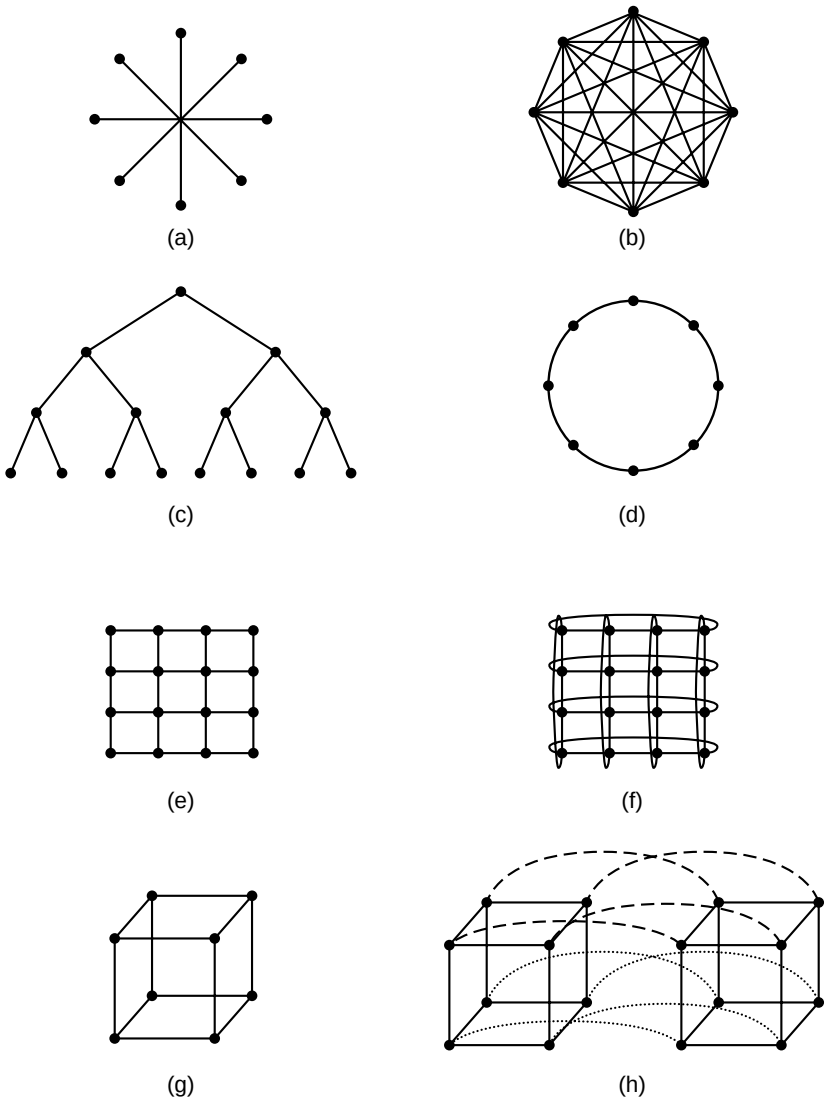


Figure 8-4. Various topologies. The heavy dots represent switches. The CPUs and memories are not shown. (a) A star. (b) A complete interconnect. (c) A tree. (d) A ring. (e) A grid. (f) A double torus. (g) A cube. (h) A 4D hypercube.

Another property of an interconnection network (or its graph) is its **diameter**. If we measure the distance between two nodes by the number of arcs that have to be traversed to get from one to the other, then the diameter of a graph is the distance between the two nodes that are the farthest apart (i.e., have the greatest distance between them). The diameter of an interconnection network is related to the

worst-case delay when sending packets from CPU to CPU or from CPU to memory because each hop across a link takes a finite amount of time. The smaller the diameter is, the better the worst-case performance is. Also important is the average distance between two nodes, since this relates to the average packet transit time.

Yet another important property of an interconnection network is its transmission capacity, that is, how much data it can move per second. One useful measure of this capacity is the **bisection bandwidth**. To compute this quantity, we first have to (conceptually) partition the network into two equal (in terms of number of nodes) but unconnected parts by removing a set of arcs from its graph. Then we compute the total bandwidth of the arcs that have been removed. There may be many different ways to partition the network into two equal parts. The bisection bandwidth is the minimum of all the possible partitions. The significance of this number is that if the bisection bandwidth is, say, 800 bits/sec, then if there is a lot of communication between the two halves, the total throughput may be limited to only 800 bits/sec, in the worst case. Many designers believe bisection bandwidth is the most important metric of in interconnection network. Many interconnection networks are designed with the goal of maximizing the bisection bandwidth.

Interconnection networks can be characterized by their **dimensionality**. For our purposes, the dimensionality is determined by the number of choices there are to get from the source to the destination. If there is never any choice (i.e., there is only one path from each source to each destination), the network is zero dimensional. If there is one dimension in which a choice can be made, for example, go east or go west, the network is one dimensional. If there are two axes, so a packet can go east or west or alternatively, go north or south, the network is two dimensional, and so on.

Several topologies are shown in Fig. 8-4. Only the links (lines) and switches (dots) are shown here. The memories and CPUs (not shown) would typically be attached to the switches by interfaces. In Fig. 8-4(a), we have a zero-dimensional **star** configuration, in which the CPUs and memories would be attached to the outer nodes, with the central one just doing switching. Although a simple design, for a large system, the central switch is likely to be a major bottleneck. Also, from a fault-tolerance perspective, this is a poor design since a single failure at the central switch completely destroys the system.

In Fig. 8-4(b), we have another zero-dimensional design that is at the other end of the spectrum, a **full interconnect**. Here every node has a direct connection to every other node. This design maximizes the bisection bandwidth, minimizes the diameter, and is exceedingly fault tolerant (it can lose any six links and still be fully connected). Unfortunately, the number of links required for k nodes is $k(k-1)/2$, which quickly gets out of hand for large k.

Yet a third zero-dimensional topology is the **tree**, illustrated in Fig. 8-4(c). A problem with this design is that the bisection bandwidth is equal to the link capacity. Since there will normally be a lot of traffic near the top of the tree, the top

few nodes will become bottlenecks. One way around this problem is to increase the bisection bandwidth by giving the upper links more bandwidth. For example, the lowest level links might have a capacity b, the next level might have a capacity $2b$ and the top-level links might each have $4b$. Such a design is called a **fat tree** and has been used in commercial multicomputers, such as the (now-defunct) Thinking Machines' CM-5.

The **ring** of Fig. 8-4(d) is a one-dimensional topology by our definition because every packet sent has a choice of going left or going right. The **grid** or **mesh** of Fig. 8-4(e) is a two-dimensional design that has been used in many commercial systems. It is highly regular, easy to scale up to large sizes, and has a diameter that only increases as the square root of the number of nodes. A variant on the grid is the **double torus** of Fig. 8-4(f), which is a grid with the edges connected. Not only is it more fault tolerant than the grid, but the diameter is also less because the opposite corners can now communicate in only two hops.

The **cube** of Fig. 8-4(g) is a regular three-dimensional topology. We have illustrated a $2 \times 2 \times 2$ cube, but in the general case it could be a $k \times k \times k$ cube. In Fig. 8-4(h) we have a four-dimensional cube constructed from two three-dimensional cubes with the corresponding edges connected. We could make a five-dimensional cube by cloning the structure of Fig. 8-4(h) and connecting the corresponding nodes to form a block of four cubes. To go to six dimensions, we could replicate the block of four cubes and interconnect the corresponding nodes, and so on. An n-dimensional cube formed this way is called a **hypercube**. Many parallel computers use this topology because the diameter grows linearly with the dimensionality. Put in other words, the diameter is the base 2 logarithm of the number of nodes, so, for example, a 10-dimensional hypercube has 1024 nodes but a diameter of only 10, giving excellent delay properties. Note that in contrast, 1024 nodes arranged as a 32×32 grid has a diameter of 62, more than six times worse than the hypercube. The price paid for the smaller diameter is that the fanout and thus the number of links (and the cost) is much larger for the hypercube. Nevertheless, the hypercube is a common choice for high-performance systems.

Switching

An interconnection network consists of switches and wires connecting them. In Fig. 8-5, we see a small network with four switches. Each switch in this example has four input ports and four output ports. In addition, each switch has some CPUs and interconnect circuitry (not shown in full). The job of the switch is to accept packets arriving on any input port and send each one out on the correct output port.

Each output port is connected to an input port belonging to another switch by a serial or parallel link, shown as dotted lines in Fig. 8-5. Serial links transfer one bit at a time. Parallel links transfer multiple bits at once and also have signals for

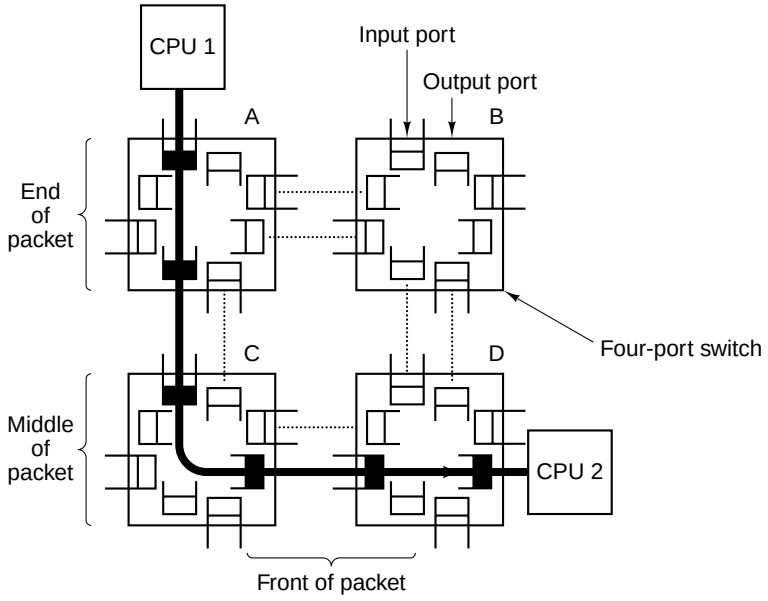


Figure 8-5. An interconnection network in the form of a four-switch square grid. Only two of the CPUs are shown.

controlling the link. Parallel links have a higher performance than serial links of the same clock rate but have the problem of **skew** (making sure all the bits arrive at same time) and are much more expensive.

Several switching strategies are possible. In one strategy, called **circuit switching**, before a packet is sent, the entire path from the source to the destination is reserved in advance. All ports and buffers are claimed, so that when transmission starts, all the necessary resources are guaranteed to be available and the bits can move at full speed from the source, through all the switches, to the destination. Figure 8-5 illustrates circuit switching, with a circuit reserved from CPU 1 to CPU 2, indicated by the heavy curved black arrow. Here three input ports and three output ports have been reserved.

Circuit switching can be compared to reserving a parade route through a city and having the police block off all side streets with barricades. It requires advance planning, but once it has been done, the marchers can proceed at full speed without interference from competing traffic. The disadvantage is that advance planning is required and competing traffic is prohibited, even when no marchers (packets) are in sight.

A second switching strategy is **store-and-forward packet switching**. Here no advance reservation is needed. Instead, the source sends a complete packet to the first switch where it is stored in its entirety. In Fig. 8-6(a) CPU 1 is the source, and the entire packet, destined for CPU 2, is first buffered inside switch A.

Once the packet has been fully accumulated in switch A, it is moved to switch C, as shown in Fig. 8-6(b). After the entire packet has fully arrived in switch C, it is moved to switch D, as shown in Fig. 8-6(c). Finally, it is sent to the destination, CPU 2. Note that no setup is required and no resources are reserved in advance.

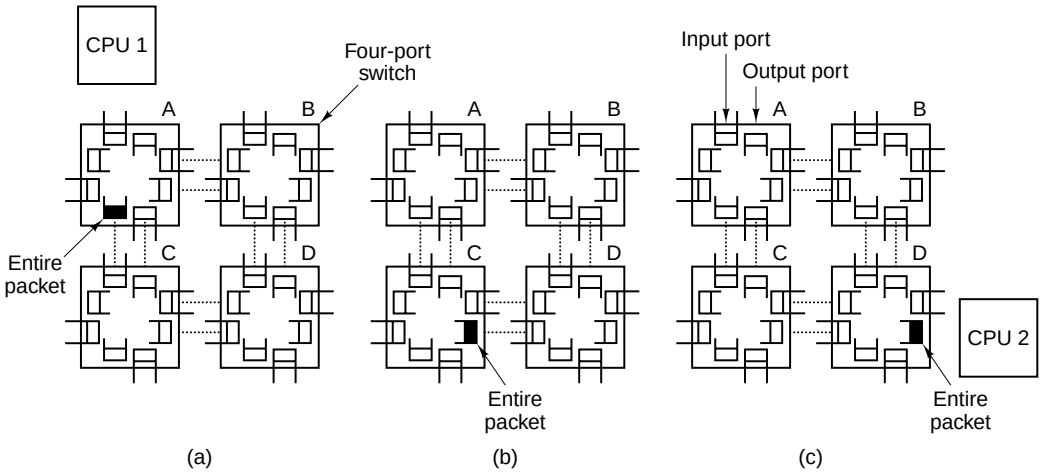


Figure 8-6. Store-and-forward packet switching.

Store-and-forward switches must buffer packets because when a source (e.g., a CPU, memory, or switch) presents a packet, the required output port may be currently busy transmitting another packet. If no buffering were present, incoming packets needing an occupied output port would have to be dropped, leading to an unreliable interconnection network. Three buffering strategies are used. In the first one, **input buffering**, one or more buffers are associated with each input port in the form of a first-in first-out queue. If the packet at the head of the queue cannot be transmitted because the output port it needs is busy, it just waits.

The trouble with this design is that if a packet is waiting for some output port, the packet behind it is stuck, even if it is destined for an idle port. This situation is called **head-of-line blocking**. It can be compared to a sequence of cars on a two-lane road that is stuck because the first car wants to turn left but cannot do so due to oncoming traffic in the other lane. Even though the second and subsequent cars want to go straight, the head car is blocking them.

Head-of-line blocking can be eliminated by **output buffering**. In this design, the buffers are associated with the output ports, not the input ports. As the bits of an incoming packet arrive, they are stored in a buffer associated with the correct output port. In this way, packets destined for port *m* cannot block packets destined for port *n*.

Both input buffering and output buffering have a fixed number of buffers associated with each port. If more packets must be stored than there is room there, packets will have to be dropped. One way to improve this situation is to use

common buffering, in which a single pool of buffers is dynamically allocated to ports as needed. However, this scheme requires a more complex administration to keep track of the buffers, and also allows one busy connection to hog all the buffers, effectively starving other connections. Also, each switch needs to be able to hold the largest packet, and probably several maximum packets, which tends to increase memory requirements and drive down the maximum packet size.

While store-and-forward packet switching is flexible and efficient, it does have the problem of increasing latency (delay) through the interconnection network. Suppose that the time to move a packet one hop in Fig. 8-6 is T nsec. Since the packet must be copied four times to get it from CPU 1 to CPU 2 (to A, to C to D, and to the destination CPU), and no copy can begin until the previous one is finished, the latency through the interconnection network is $4T$. One way out is to design a hybrid network, with some of the properties of circuit switching and some of the properties of packet switching. For example, each packet can be logically divided into smaller units. As soon as the first unit arrives at a switch, it can be moved to the next switch, even before the tail of the packet has arrived.

This approach differs from circuit switching in that no resources are reserved end-to-end in advance. Consequently contention for resources (ports and buffers) is possible. In **virtual cut through routing**, when the first unit of a packet cannot move, the rest of the packet continues to pour in. In the worst case, this scheme degrades to store-and-forward packet switching. In an alternative approach, **wormhole routing**, when the first unit cannot go forward, the source is told to stop transmitting, so the packet may end up being strung out over two or possibly even more switches like a worm. When the necessary resources become available, the packet can go forward.

It is worth noting that both of these approaches to switching use something analogous to pipelining instructions on a CPU. At any instant, each switch is only doing a small fraction of the work, but together they achieve a higher performance than any one could do alone.

Routing Algorithms

In any network with a dimensionality of one or more, choices have to be made about how to route packets from the source to the destination. Often multiple routes exist. The rule that determines which sequence of nodes a packet must follow from the source to the destination is called the **routing algorithm**.

Good routing algorithms are needed because often multiple paths are available. A good routing algorithm can spread the load over multiple links in order to fully utilize the available bandwidth. Furthermore, the routing algorithm must avoid deadlock within the interconnection network. A **deadlock** occurs when multiple packets in transit at the same time have claimed resources in such a way that none of them can make any forward progress and all will remained blocked forever.

An example of a deadlock in a circuit-switched interconnection network is given in Fig. 8-7. (Deadlock can also occur in packet-switched networks, but it is more graphic in circuit-switched ones.) Here each CPU is trying to send a packet to the CPU diagonally opposite it. Each one has managed to reserve the input and output ports on its local switch, as well as one input port on the next switch, but it is unable to get the necessary output port on the second switch, so it just waits until that port is available. Unfortunately, if all four CPUs start this process simultaneously, all of them will block and the network will hang forever.

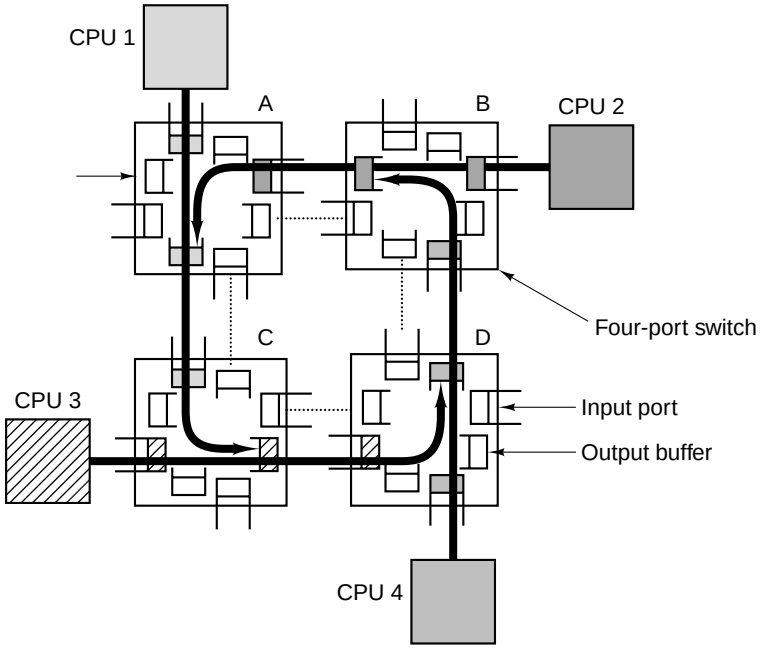


Figure 8-7. Deadlock in a circuit-switched interconnection network.

Routing algorithms can be categorized as source routing or distributed routing. In **source routing**, the source determines the full route through the interconnection network in advance, expressed as a list of port numbers to be used at each switch along the way. Typically if the path passes through *k* switches, the first *k* bytes of each packet will contains the *k* output port numbers required, 1 byte per port. When the packet reaches a switch, the first byte is stripped off and used to determine the output port to use. The remaining portion of the packet is then routed to the correct port. At each hop along the path, the packet gets 1 byte shorter, exposing a new port number to be selected next.

In distributed routing, each switch makes a decision about which port to send each arriving packet to. If it makes the same decision for each packet headed to a given destination, the routing is said to be **static**. If it takes the current traffic into account, the routing is said to be **adaptive**.

One popular routing algorithm for rectangular grids in any number of dimensions that is known to be deadlock free is **dimensional routing**. In this algorithm, the packet is first moved along the *x*-axis to the correct coordinate, then along the *y*-axis to the correct coordinate, and so on for higher dimensions. For example, to go from (3, 7, 5) to (6, 9, 8), the packet would first go from $x = 3$ to $x = 6$ through (4, 7, 5), (5, 7, 5), and (6, 7, 5). Then it would work on the *y*-axis by going to (6, 8, 5) and (6, 9, 5). Finally, it would go along the *z*-axis to (6, 9, 6), (6, 9, 7) and (6, 9, 8). This algorithm prevents the cycles that cause a deadlock.

8.1.3 Performance

The point of building a parallel computer is to make it go faster than a uniprocessor machine. If it does not achieve that simple goal, it is not worth having. Furthermore, it should achieve the goal in a cost-effective manner. A machine that is twice as fast as a uniprocessor at 50 times the cost is not likely to be a big seller. In this section we will examine some of the performance issues associated with parallel computer architectures.

Hardware Metrics

From a hardware perspective, the performance metrics of interest are the CPU and I/O speeds and the performance of the interconnection network. The CPU and I/O speeds are the same as in the uniprocessor case, so the key parameters of interest in a parallel system are those associated with the interconnect. There are two key items: latency and bandwidth, which we will now look at in turn.

The roundtrip latency is the time it takes for a CPU to send a packet and get a reply. If the packet is sent to a memory, then the latency measures the time to read or write a word or block of words. If it is sent to another CPU, it measures the interprocessor communication time for packets of that size. Usually, the latency of interest is for minimal packets, often one word or a small cache line.

The latency is built up from several factors, and is different for circuit-switched, store-and-forward, virtual cut through, and wormhole-routed interconnects. For circuit switching, the latency is the sum of the setup time and the transmission time. To set up a circuit, a probe packet has to be sent out to reserve the resources and then report back. Once that has happened, the data packet has to be assembled. When it is ready, bits can flow at full speed, so if the total setup time is T_s, the packet size is p bits, and the bandwidth b bits/sec, the one-way latency is $T_s + p/b$. If the circuit is full duplex, then there is no setup time for the reply, so the minimum latency for sending a p bit packet and getting a p bit reply is $T_s + 2p/b$ sec.

For packet switching, it is not necessary to send a probe packet to the destination in advance, but there is still some internal setup time to assemble the packet, T_a. Here the one-way transmission time is $T_a + p/b$, but this is only the time to

get the packet into the first switch. There is a finite delay within the switch, say T_d and then the process is repeated to the next switch and so on. The T_d delay is composed of both processing time and queueing delay, waiting for the output port to become free. If there are n switches, then the total one-way latency is $T_a + n(p/b + T_d) + p/b$, where the final term is due to the copy from the last switch to the destination.

The one-way latencies for virtual cut through and wormhole routing in the best case are close to $T_a + p/b$ because there is no probe packet to set up a circuit, and no store-and-forward delay either. Basically, it is the initial setup time to assemble the packet, plus the time to push the bits out the door. In all cases, propagation delay has to be added, but that is usually small.

The other hardware metric is bandwidth. Many parallel programs, especially in the natural sciences, move a lot of data around, so the number of bytes/sec that the system can move is critical to performance. Several metrics for bandwidth exist. We have seen one of them—bisection bandwidth—already. Another one is the **aggregate bandwidth**, which is computed by simply adding up the capacities of all the links. This number gives the maximum number of bits that can be in transit at once. Yet another important metric is the average bandwidth out of each CPU. If each CPU is capable of outputting 1 MB/sec, it does little good that the interconnect has a bisection bandwidth of 100 GB/sec. Communication will be limited by how much data each CPU can output.

In practice, actually achieving anything even close to the theoretical bandwidth is very difficult. Many sources of overhead work to reduce the capacity. For example, there is always some per-packet overhead associated with each packet: assembling it, building its header, and getting it going. Sending 1024 4-byte packets will never achieve the same bandwidth as sending 1 4096-byte packet. Unfortunately, for achieving low latencies, using small packets is better, since large ones block the lines and switches too long. Thus there is an inherent conflict between achieving low average latencies and high-bandwidth utilization. For some applications, one is more important than the other and for other applications it is the other way around. It is worth noting, however, that you can always buy more bandwidth (by putting in more or wider wires), but you cannot buy lower latencies. Thus it is generally better to err on the side of making latencies as short as possible, and worry about bandwidth later.

Software Metrics

Hardware metrics like latency and bandwidth look at what the hardware is capable of doing. However, users have a different perspective. They want to know how much faster their programs are going to run on a parallel computer than on a uniprocessor. For them, the key metric is speedup: how much faster a program runs on an n-processor system than on a 1-processor system. Typically these results are shown in graphs like those of Fig. 8-8. Here we see several

different parallel programs run on a multicomputer consisting of 64 Pentium Pro CPUs. Each curve shows the speedup of one program with k CPUs as a function of k. Perfect speedup is indicated by the dotted line, in which using k CPUs makes the program go k times faster, for any k. Few programs achieve perfect speedup, but some come close. The N-body problem parallelizes extremely well; awari (an African board game) does reasonably well; but inverting a certain skyline matrix does not go more than five times faster no matter how many CPUs are available. The programs and results are discussed in (Bal et al., 1998).

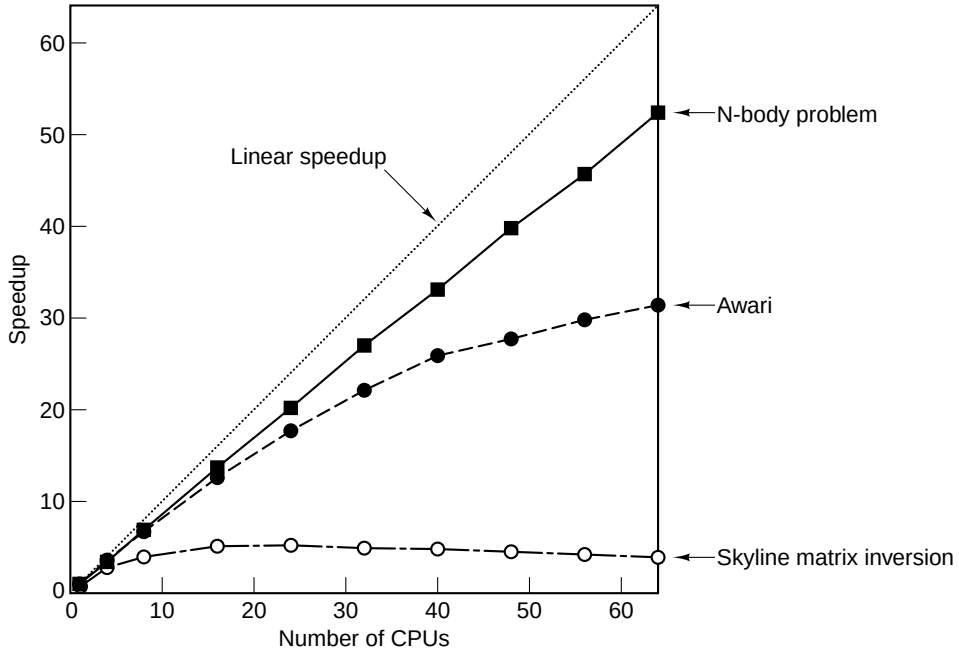


Figure 8-8. Real programs achieve less than the perfect speedup indicated by the dotted line.

Part of the reason that perfect speedup is nearly impossible to achieve is that almost all programs have some sequential component, often the initialization phase, reading in the data, or collecting the results. Having many CPUs does not help here. Suppose that a program runs for T sec on a uniprocessor, with a fraction f of this time being sequential code and a fraction $(1 - f)$ being potentially parallelizable, as shown in Fig. 8-9(a). If the latter code can be run on n CPUs with no overhead, its execution time can be reduced from $(1 - f)T$ to $(1 - f)T/n$ at best, as shown in Fig. 8-9(b). This gives a total execution time for the sequential and parallel parts of $fT + (1 - f)T/n$. The speedup is just the execution time of the original program, T, divided by this new execution time:

$$Speedup = \frac{n}{1 + (n - 1)f}$$

For $f = 0$ we can get linear speedup, but for $f > 0$, perfect speedup is not possible due to the sequential component. This result is known as **Amdahl's law**.

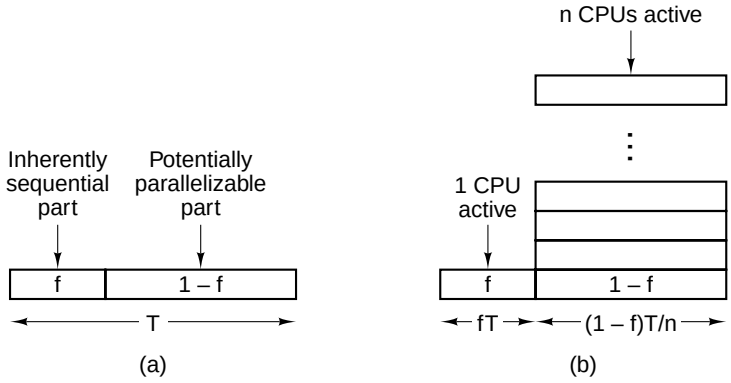


Figure 8-9. (a) A program has a sequential part and a parallelizable part. (b) Effect of running part of the program in parallel.

Amdahl's law is not the only reason perfect speedup is nearly impossible to achieve. Nonzero communication latencies, finite communication bandwidths, and algorithmic inefficiencies can also play a role. Also, even if 1000 CPUs were available, not all programs can be written to make use of so many CPUs, and the overhead in getting them all started may be significant. Furthermore, often the best-known algorithm does not parallelize well, so a suboptimal algorithm must be used in the parallel case. This all said, for many applications, having the program run n times faster is highly desirable, even if it takes $2n$ CPUs to do it. CPUs are not that expensive, after all, and many companies live with considerably less than 100% efficiency in other parts of their businesses.

Achieving High Performance

The most straightforward way to improve performance is to add more CPUs to the system. However, this addition must be done in such a way as to avoid creating any bottlenecks. A system in which one can add more CPUs and get correspondingly more computing power is said to be **scalable**.

To see some of the implications of scalability, consider four CPUs connected by a bus, as illustrated in Fig. 8-10(a). Now imagine scaling the system to 16 CPUs by adding 12 more, as shown in Fig. 8-10(b). If the bandwidth of the bus is b MB/sec, then by quadrupling the number of CPUs, we have also reduced the available bandwidth per CPU from $b/4$ MB/sec to $b/16$ MB/sec. Such a system is not scalable.

Now we do the same thing with a grid-based system, as shown in Fig. 8-10(c) and Fig. 8-10(d). With this topology, adding new CPUs also adds new links, so

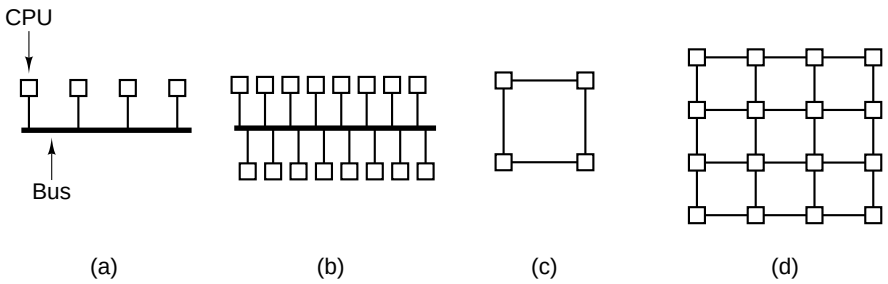


Figure 8-10. (a) A 4-CPU bus-based system. (b) A 16-CPU bus-based system. (c) A 4-CPU grid-based system. (d) A 16-CPU grid-based system.

scaling the system up does not cause the aggregate bandwidth per CPU to drop, as it does with a bus. In fact, the ratio of links to CPUs increases from 1.0 with 4 CPUs (4 CPUs, 4 links) to 1.5 with 16 CPUs (16 CPUs, 24 links), so adding CPUs improves the aggregate bandwidth per CPU.

Of course, bandwidth is not the only issue. Adding CPUs to the bus does not increase the diameter of the interconnection network or latency in the absence of traffic, whereas adding them to the grid does. For an $n \times n$ grid, the diameter is $2(n - 1)$, so the worst (and average) case latency increases roughly as the square root of the number of CPUs. For 400 CPUs, the diameter is 38, whereas for 1600 CPUs it is 78, so quadrupling the number of CPUs approximately doubles the diameter and thus the average latency.

Ideally, a scalable system should maintain the same average bandwidth per CPU and a constant average latency as CPUs are added. In practice, keeping enough bandwidth per CPU is doable, but in all practical designs, latency grows with size. Having it grow logarithmically, as in a hypercube, is about the best that can be done.

The problem with having latency grow as the system scales up is that latency is often fatal to performance in fine- and medium-grained applications. If a program needs data that are not in its local memory, there is often a substantial delay in getting them, and the bigger the system, the longer the delay, as we have just seen. This problem is equally true of multiprocessors as multicomputers, since in both cases the physical memory is invariably divided up into far-flung modules.

As a consequence of this observation, system designers often go to great lengths to reduce, or at least hide, the latency, using several techniques we will now mention. The first latency-hiding technique is data replication. If copies of a block of data can be kept at multiple locations, accesses from those locations can be speeded up. One such replication technique is caching, in which one or more copies of data blocks are kept close to where they are being used, as well as where they "belong." However, another strategy is to maintain multiple peer copies—copies that have equal status—as opposed to the asymmetric primary/secondary

relationship used in caching. When multiple copies are maintained, in whatever form, key issues are where the data blocks are placed, when, and by whom. Answers range from dynamic placement on demand by the hardware, to intentional placement at load time following compiler directives. In all cases, managing consistency is an issue.

A second technique for hiding latency is **prefetching**. If a data item can be fetched before it is needed, the fetching process can be overlapped with normal execution, so that when the item is needed, it will be there. Prefetching can be automatic or under program control. When a cache loads not only the word being referenced, but an entire cache line containing the word, it is gambling that the succeeding words are also likely to be needed soon.

Prefetching can also be controlled explicitly. When the compiler realizes that it will need some data, it can put in an explicit instruction to go get them, and put that instruction sufficiently far in advance that the data will be there in time. This strategy requires that the compiler have a complete knowledge of the underlying machine and its timing, as well as control over where all data are placed. Such speculative LOAD instructions work best when it is known for sure that the data will be needed. Getting a page fault on a LOAD for a path that is ultimately not taken is very costly.

A third technique that can hide latency is **multithreading**. Most modern systems support the concept of multiprogramming, in which multiple processes can run simultaneously (or in pseudoparallel on a timesharing basis). If switching between processes can be made fast enough, for example, by giving each one its own memory map and hardware registers, then when one process blocks waiting for remote data to arrive, the hardware can quickly switch to another one that is able to continue. In the limiting case, the CPU runs the first instruction from thread one, the second instruction from thread two, and so on. In this way, the CPU can be kept busy, even in the face of long memory latencies for the individual threads.

In fact, some machines automatically cycle among the processes, switching after every instruction, to hide long latencies. One of the first supercomputers, the CDC 6600, carried this idea to such an extreme that it was advertised as having 10 peripheral processing units that could run in parallel. In reality there was only one peripheral processor, which simulated 10 of them by running instructions from each one in round robin order, first one instruction from peripheral processor 1, then one instruction from peripheral processor 2, and so on.

A fourth technique for hiding latency is using nonblocking writes. Normally, when a STORE instruction is executed, the CPU waits until the STORE has completed before continuing. With nonblocking writes, the memory operation is started, but the program just continues anyway. Continuing past a LOAD is harder, but with out-of-order execution, even that is possible.

8.1.4 Software

Although this chapter is primarily about parallel computer architectures, a few words about software are appropriate too. After all, without parallel software, parallel hardware is of little use, so good hardware designers take the needs of the software into account when designing the hardware. For a discussion about software for parallel computers, see (Wilkinson and Allen, 1999).

There are four general approaches to producing software for parallel computers. At one extreme is the addition of special numerical libraries to otherwise normal sequential languages. For example, a library procedure to invert a large matrix or solve a set of partial differential equations could be called from a sequential program and do its work on a parallel processor without the programmer even being aware of the existence of parallelism. The trouble with this approach is that parallelism can only be used in a few procedures and the bulk of the code will remain sequential.

A second approach is the addition of special libraries containing communication and control primitives. Here the programmer is responsible for creating and managing the parallelism within a conventional programming language using these additional primitives.

The next step up is to add a few special constructs to existing programming languages, such as the ability to fork off new parallel processes easily, execute the iterations of a loop in parallel, or do arithmetic on all the elements of a vector at the same time. This approach is widely used and numerous programming languages have been modified to include some parallelism.

The most extreme approach is to invent an entirely new language especially for parallel processing. The obvious advantage of inventing a new language is that such a language is well suited for parallel programming, but the equally obvious disadvantage is that programmers must learn a new language. Quite a few of them have something else to do with their time. Most new parallel languages are imperative (with instructions to modify state variables), but a few are functional, logic based, or object oriented.

There have been so many libraries, extensions, and new languages invented for parallel programming, and they span such a broad spectrum of possibilities, that it is impossible to classify all of them in any reasonable way. Instead of even trying, we will focus on five key issues that form the heart of all software for parallel computers:

1. Control models.
2. Granularity of parallelism.
3. Computational paradigms.
4. Communication methods.
5. Synchronization primitives.

Each of these will now be discussed in turn.

Control Models

Probably the most fundamental choice the software must make is whether there will be one thread of control or multiple threads of control. In the former model, there is one program and one program counter, but multiple sets of data. As each instruction is issued, it is carried out on all data sets simultaneously, by different processing elements.

As an example, imagine a weather program with hourly temperature measurements from thousands of remote sensors that have to be averaged for each sensor. When the program issues the instruction

```
LOAD THE TEMPERATURE FOR 1 A.M. INTO REGISTER R1
```

each processor carries it out, using its own data and its own R1. Later, when the program issues the instruction

```
ADD THE TEMPERATURE FOR 2 A.M. TO REGISTER R1
```

again, each processor does it using its own data. At the end of the calculation, each processor will have computed the mean temperature for a different sensor.

The implications for this programming model for the hardware are enormous. It effectively says that each processing element is basically an ALU and a memory, with no instruction decoding logic of its own. Instead a single centralized unit fetches instructions and tells the ALUs what to do next.

In the alternative model, multiple threads of control exist, each one with its own program counter, registers, and local variables. Each thread of control executes its own program on its own data, possibly communicating or synchronizing with other threads of control from time to time. Many variations on this basic idea exist, and together they form the dominant model for parallel processing. For this reason, we will now focus primarily on parallel computing involving multiple threads of control.

Granularity of Parallelism

Control parallelism can be introduced at various levels. At the lowest level, individual machine instructions can contain parallelism (e.g., the IA-64). At this level, programmers are usually not aware of the parallelism; it is managed by the compiler or hardware.

One level up we come to **block-level parallelism**, which allows programmers explicit control over which statements are to be executed sequentially and which in parallel. One of the most elegant ways of expressing this kind of parallelism is due to Algol 68, in which

begin Statement-1; Statement-2; Statement-3 **end**

was used to create a block with (for example) three arbitrary statements to be executed sequentially. In contrast,

begin Statement-1, Statement-2, Statement-3 **end**

was used to execute the same three statements in parallel. By careful placement of semicolons, commas, parentheses, and **begin**/**end** delimiters, arbitrary combinations of sequential and parallel execution could be expressed.

A somewhat coarser grain of parallelism is present when it is possible to call a procedure and not have the caller wait for it before continuing. Not waiting means that the caller and callee will run in parallel. If the caller is in a loop that calls a procedure on every iteration and waits for none of them, then a large number of parallel procedures can be started at once.

Another form of parallelism is to have a method for a process to create or fork off multiple **threads** or **lightweight processes**, all of which run within the process address space. Each thread has its own program counter, registers, and stack but otherwise shares all the rest of the address space (and all the global variables) with all the other threads. (In contrast to threads, different processes do not share a common address space.) Threads run independently of one another, possibly on different CPUs. In some systems, the operating system is aware of all the threads and does the thread scheduling. In other systems, each user process does its own thread scheduling and management, without the operating system even knowing about the threads.

Finally, the coarsest form of parallelism is having multiple independent processes work together to solve a problem. Unlike threads, which share a common address space, independent processes do not, so they must cooperate at arms' length, which means that the problem must be divided into relatively large chunks, one for each process. However, independent processes also offer the most opportunity for exploiting large-scale parallelism especially on multicomputers.

Computational Paradigms

Most parallel programs, especially those involving large numbers of threads or independent processes, use some underlying paradigm to structure their work. Many such paradigms exist. In this section we will just mention a few of the more popular ones.

A generalization of the idea of having a parallel program consist of one thread of control and multiple execution units is the **SPMD** (**Single Program Multiple Data**) paradigm. The idea here is that although the system consists of multiple independent processes, they all run the same program, but on different data sets. Only now, unlike in the temperature example, they are not in lockstep, synchronized down to the last instruction. Instead, each one performs the same

computation, but at its own pace.

A second paradigm is the **pipeline**, illustrated in Fig. 8-11(a) with three processes. Data are fed into the first process, which transforms them and outputs them for the second process to read, and so on. If the data stream is long (e.g., a video presentation), all the processors may be busy at the same time. UNIX pipelines work like this and can be run as separate processes in parallel on a multicomputer or multiprocessor.

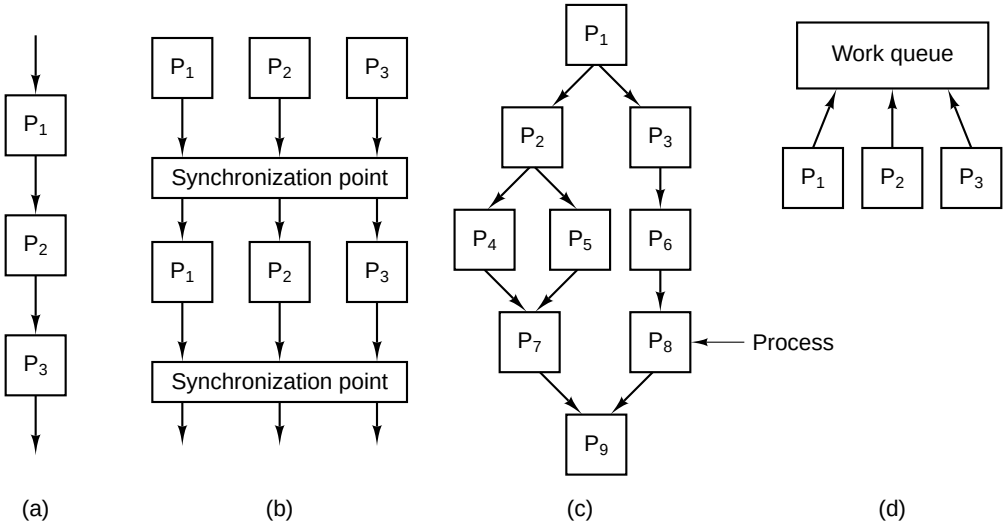


Figure 8-11. Computational paradigms. (a) Pipeline. (b) Phased computation. (c) Divide and conquer. (d) Replicated worker.

Another paradigm is the computation of Fig. 8-11(b), in which the work is divided up into phases, for example, iterations of a loop. During each phase, multiple processes work in parallel, but when each one has finished, it waits until all the others are done too before starting the next phase. A fourth paradigm is **divide and conquer**, depicted in Fig. 8-11(c), in which one process starts off and then forks off other processes to which it can pass some of the work. An analogy in the construction world is a general contractor who gets an order, then subcontracts out much of the work to masonry, electrical, plumbing, painting, and other subcontractors. These, in turn, may contract out some of their work to yet more specialized subcontractors.

Our last model is the **replicated worker paradigm**, sometimes called a **task farm**, and illustrated in Fig. 8-11(d). Here, a centralized work queue exists and workers take tasks out of the queue and complete them. If a task generates new tasks, these are added to the central queue. Whenever a worker finishes its current task, it goes to the task queue to get a new one. This paradigm can also be implemented with an active manager at the top handing out work, rather than

having the workers forage for it on their own.

Communication Methods

When a program is split up into pieces, say, processes, which run in parallel, the pieces (processes) often need to communicate with one another. This communication can be done in one of two ways: shared variables or explicit message passing. In the former method, all the processes have access to the common logical memory and can communicate by reading and writing it. For example, one process can set a variable and another one, in a different process, can read it.

On a multiprocessor, variables can be shared among multiple processes by mapping the same page into each process' address space. Then shared variables can be read and written using ordinary LOAD and STORE machine instructions. However, even on a multicomputer, with no shared physical memory, logical sharing of variables is still possible, albeit slightly more complicated. As we saw above, Linda supports the notation of a shared tuple space, even on a multicomputer, and Orca supports the concept of shared objects across machine boundaries. Sharing a single address space on a multicomputer and paging over the interconnection network is also a possibility. In short, allowing processes to communicate through shared variables is possible on both multiprocessors and multicomputers.

The alternative to communication via shared memory is communication via explicit message passing. In this model, processes use primitives such as send and receive to communicate. One process does a send, naming another process as the destination. As soon as the latter does a receive, the message is copied into the receiver's address space.

Message passing has many variations, but all of them come down to having two primitives send and receive which are used to transmit messages. These are generally implemented as system calls. We will go into some of the details later in this chapter.

Another issue in message passing is how many receivers there are. The simplest case is one sender and one receiver, known as **point-to-point message passing**. However, sometimes it is useful for a message to be delivered to all processes, called **broadcasting**, or to a specific subset of the processes, known as **multicasting**.

It should be noted that message passing is easy to implement on a multiprocessor by simply copying from the sender to the receiver. Thus the issues of physical shared memory (multiprocessor versus multicomputer) and logical shared memory (communication via shared variables versus explicit message passing) are separate. All four combinations make sense and can be implemented. They are listed in Fig. 8-12.

Physical (hardware)	Logical (software)	Examples
Multiprocessor	Shared variables	Image processing as in Fig. 8-1
Multiprocessor	Message passing	Message passing simulated with buffers in memory
Multicomputer	Shared variables	DSM, Linda, Orca, etc. on an SP/2 or a network of PCs
Multicomputer	Message passing	PVM or MPI on an SP/2 or a network of PCs

Figure 8-12. Combinations of physical and logical sharing.

Synchronization Primitives

Parallel processes not only need to communicate, they also often need to synchronize their actions. An example of synchronization occurs when processes logically share variables and have to make sure that while one process is writing to a shared data structure, no other process is attempting to read from it. In other words, some form of **mutual exclusion** is needed to prevent multiple processes from using the same data at the same time.

Various software primitives that can be used to ensure mutual exclusion exist. These include semaphores, locks, mutexes, and critical sections. What they all have in common is that they allow a process to request exclusive use of some resource (shared variable, I/O device, etc.). If permission is granted, the process can use the resource. If a second process asks for permission while the first one is still using the resource, it will be denied permission or blocked until the first process has released the resource.

A second kind of synchronization primitive needed in many parallel programs is some way to allow all the processes to block until a certain phase of the work is done, as shown in Fig. 8-11(b). A common primitive here is the **barrier**. When a process hits a barrier, it is blocked until all the processes have hit the barrier. When the last one arrives, all the processes are simultaneously released and allowed to continue.

8.1.5 Taxonomy of Parallel Computers

Although much more could be said about software for parallel computers, space limitations require us to get back to our main topic for this chapter, the architecture of parallel computers. Many kinds of parallel computers have been proposed and built over the years, so it is natural to ask if there is some way of categorizing them into a taxonomy. Many researchers have tried, with mixed results (Flynn, 1972; Gajski and Pier, 1985; and Treleaven, 1985). Unfortunately, the Carolus Linnaeus† of parallel computing is yet to emerge. The only scheme

† Carolus Linnaeus (1707-1778) was the Swedish biologist who devised the system now used for

that is used much is Flynn's, and even his is, at best, a very crude approximation. It is given in Fig. 8-13.

Instruction streams	Data streams	Name	Examples
1	1	SISD	Classical Von Neumann machine
1	Multiple	SIMD	Vector supercomputer, array processor
Multiple	1	MISD	Arguably none
Multiple	Multiple	MIMD	Multiprocessor, multicomputer

Figure 8-13. Flynn's taxonomy of parallel computers.

Flynn's classification is based on two concepts—instruction streams and data streams. An instruction stream corresponds to a program counter. A system with *n* CPUs has *n* program counters, hence *n* instruction streams.

A data stream consists of a set of operands. The temperature computation example given earlier has multiple data streams, one for each sensor.

The instruction and data streams are, to some extent, independent, so four combinations exist, as listed in Fig. 8-13. SISD is just the classical, sequential von Neumann computer. It has one instruction stream, one data stream, and does one thing at a time. SIMD machines have a single control unit that issues one instruction at a time, but they have multiple ALUs to carry it out on multiple data sets simultaneously. The ILLIAC IV (Fig. 2-8) is the prototype of SIMD machines. Modern SIMD machines exist and are used for scientific computing.

MISD machines are a somewhat strange category, with multiple instructions operating on the same piece of data. It is not clear if any such machines exist, although some people regard pipelined machines as MISD.

Finally, we have MIMD, which are just multiple independent CPUs operating as part of a larger system. Most parallel processors fall into this category. Both multiprocessors and multicomputers are MIMD machines.

Flynn's taxonomy stops here, but we have extended it in Fig. 8-14. SIMD has been split into two subgroups. The first one is for numeric supercomputers and other machines that operate on vectors, performing the same operation on each vector element. The second one is for parallel-type machines, such as the ILLIAC IV, in which a master control unit broadcasts instructions to many independent ALUs.

In our taxonomy, the MIMD category has been split into multiprocessors (shared-memory machines), and multicomputers (message-passing machines). Three kinds of multiprocessors exist, distinguished by the way the shared memory is implemented on them. They are called **UMA** (**Uniform Memory Access**), **NUMA** (**NonUniform Memory Access**), and **COMA** (**Cache Only Memory**

---

classifying all plants and animals into kingdom, phylum, class, order, family, genus, and species.

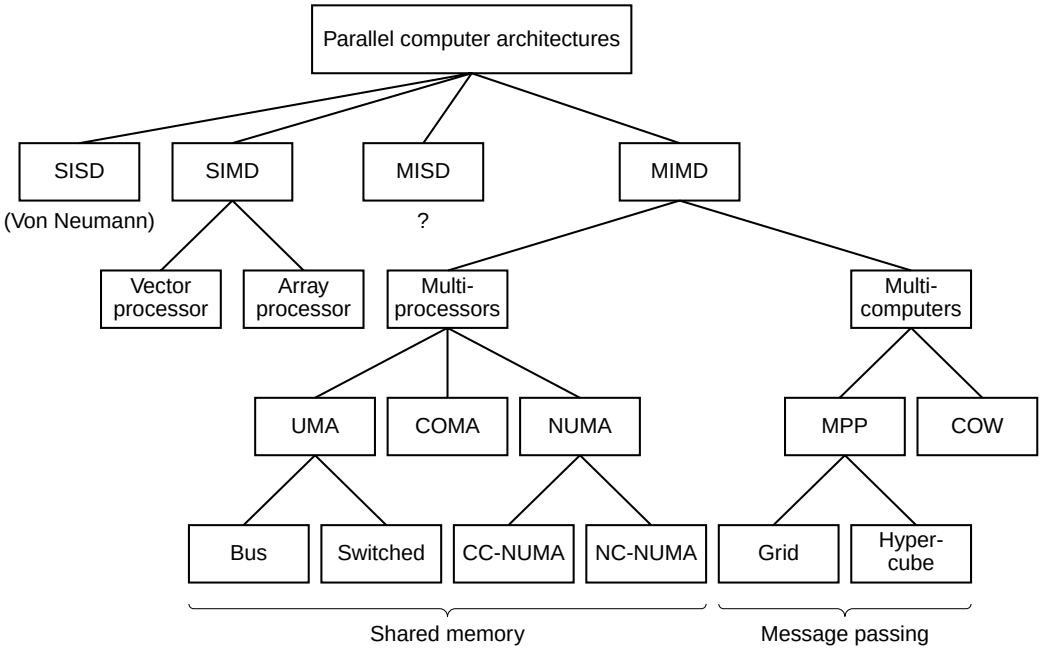


Figure 8-14. A taxonomy of parallel computers.

Access). These categories exist because in large multiprocessors, the memory is usually split up into multiple modules. UMA machines have the property that each CPU has the same access time to every memory module. In other words, every memory word can be read as fast as every other memory word. If this is technically impossible, the fastest references are slowed down to match the slowest ones, so programmers do not see the difference. This is what ''uniform'' means here. This uniformity makes the performance predictable, an important factor for writing efficient code.

In contrast, in a NUMA multiprocessor, this property does not hold. Often there is a memory module close to each CPU and accessing that memory module is much faster than accessing distant ones. The result is that for performance reasons, it matters where code and data are placed. COMA machines are also nonuniform, but in a different way. We will study each of these types and their subcategories in detail later.

The other main category of MIMD machines consists of the multicomputers, which, unlike the multiprocessors, do not have shared primary memory at the architectural level. In other words, the operating system on a multicomputer CPU cannot access memory attached to a different CPU by just executing a LOAD instruction. It has to send an explicit message and wait for an answer. The ability of the operating system to read a distant word by just doing a LOAD is what

distinguishes multiprocessors from multicomputers. As we mentioned before, even on a multicomputer, user programs may have the ability to access remote memory by using LOAD and STORE instructions, but this illusion is supported by the operating system, not the hardware. This difference is subtle, but very important. Because multicomputers do not have direct access to remote memory, they are sometimes called **NORMA** (**NO Remote Memory Access**) machines.

Multicomputers can be roughly divided into two categories. The first category contains the **MPP**s (**Massively Parallel Processors**), which are expensive supercomputers consisting of many CPUs tightly coupled by a high-speed proprietary interconnection network. The Cray T3E and IBM SP/2 are well-known examples.

The other category consists of regular PCs or workstations, possibly rack-mounted, and connected by commercial off-the-shelf interconnection technology. Logically, there is not much difference, but huge supercomputers costing many millions of dollars are used differently than networks of PCs assembled by the users for a fraction of the price of an MPP. These home-brew machines go by various names, including **NOW** (**Network of Workstations**) and **COW** (**Cluster of Workstations**).

In the following sections, we will look at the main categories, SIMD, MIMD/multiprocessors, and MIMD/multicomputers in some detail. The goal is to give a good overview of what each type is like, what the subcategories are, and what the key design principles are. Several examples will be used for illustration.

A

BINARY NUMBERS

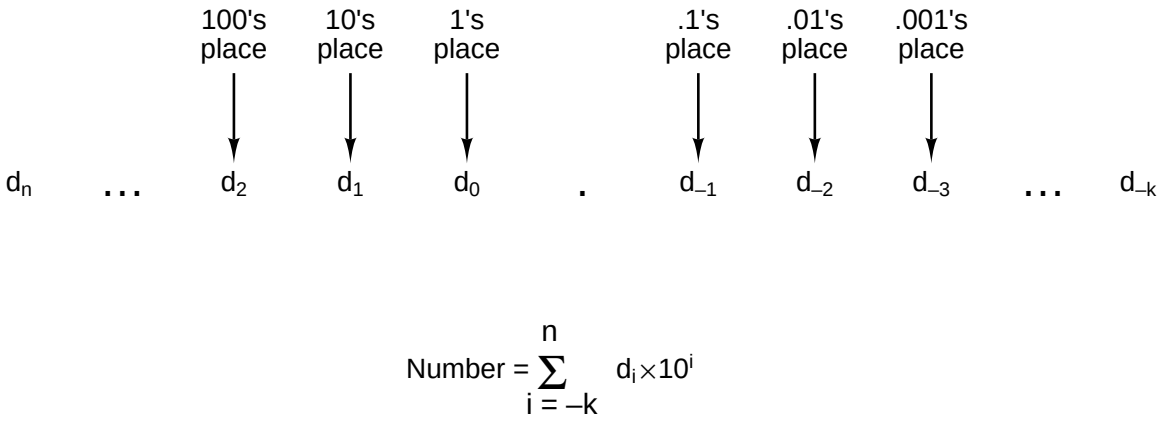


Figure A-1. The general form of a decimal number.

Binary 1 1 1 1 1 0 1 0 0 0 1

$1 \times 2^{10} + 1 \times 2^{9} + 1 \times 2^{8} + 1 \times 2^{7} + 1 \times 2^{6} + 0 \times 2^{5} + 1 \times 2^{4} + 0 \times 2^{3} + 0 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0}$

1024 + 512 + 256 + 128 + 64 + 0 +16 + 0 + 0 + 0 + 1

Octal 3 7 2 1

$3 \times 8^{3} + 7 \times 8^{2} + 2 \times 8^{1} + 1 \times 8^{0}$

1536 + 448 + 16 + 1

Decimal 2 0 0 1

$2 \times 10^{3} + 0 \times 10^{2} + 0 \times 10^{1} + 1 \times 10^{0}$

2000 + 0 + 0 + 1

Hexadecimal 7 D 1 .

$7 \times 16^{2} + 13 \times 16^{1} + 1 \times 16^{0}$

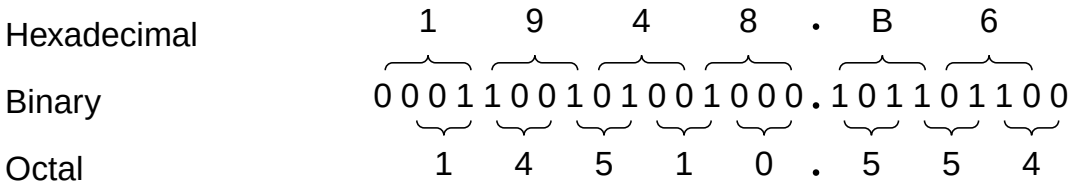
1792 + 208 + 1

Figure A-2. The number 2001 in binary, octal, and hexadecimal.

Decimal	Binary	Octal	Hex
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	3	3
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BA

Figure A-3. Decimal numbers and their binary, octal, and hexadecimal equivalents.

Example 1



Example 2

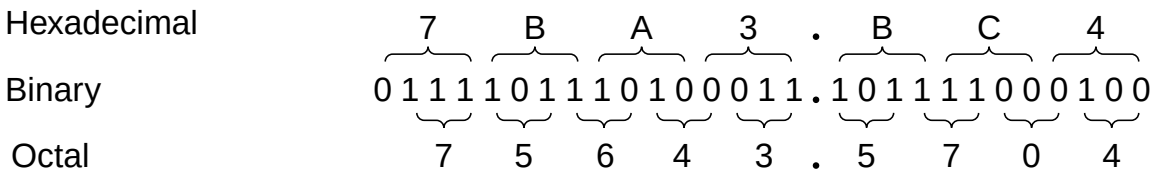


Figure A-4. Examples of octal-to-binary and hexadecimal-to-binary conversion.

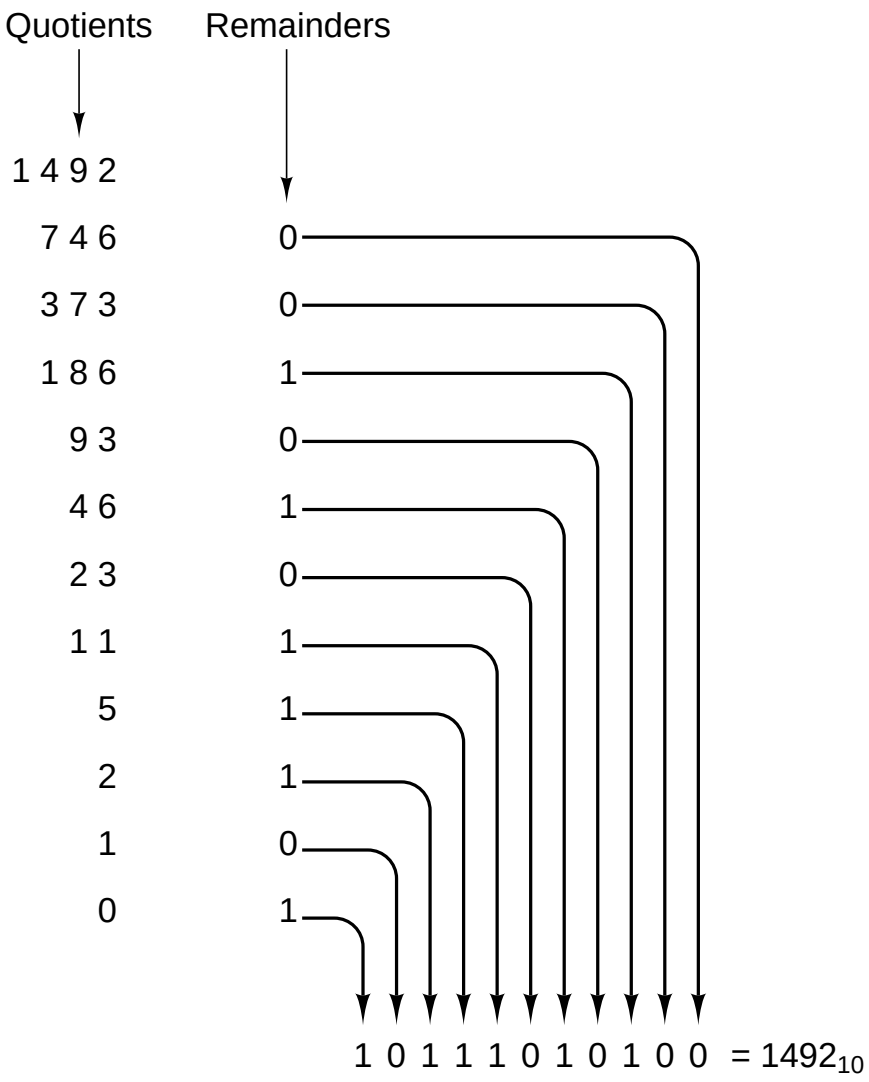


Figure A-5. Conversion of the decimal number 1492 to binary by successive halving, starting at the top and working downward. For example, 93 divided by 2 yields a quotient of 46 and a remainder of 1, written on the line below it.

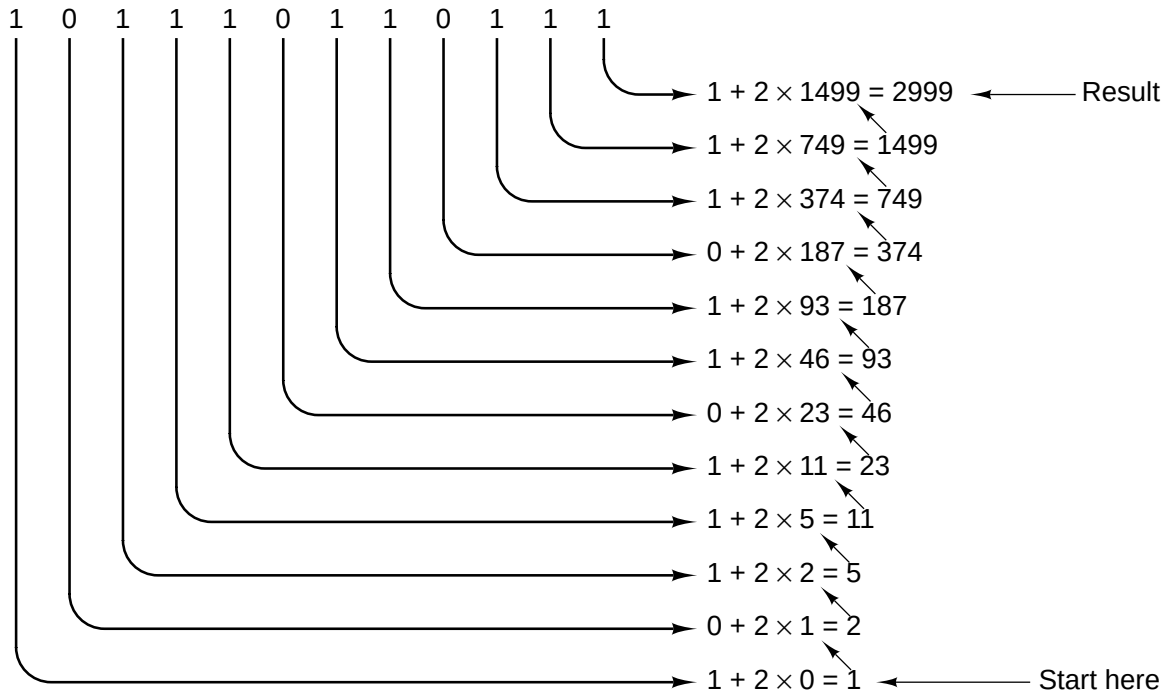


Figure A-6. Conversion of the binary number 101110110111 to decimal by successive doubling, starting at the bottom. Each line is formed by doubling the one below it and adding the corresponding bit. For example, 749 is twice 374 plus the 1 bit on the same line as 749.

N decimal	N binary	–N signed mag.	–N 1's compl.	–N 2's compl.	–N excess 128
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11011010	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Nonexistent	Nonexistent	Nonexistent	10000000	00000000

Figure A-7. Negative 8-bit numbers in four systems.

Addend	0	0	1	1
Augend	+0	+1	+0	+1
Sum	0	1	1	0
Carry	0	0	0	1

Figure A-8. The addition table in binary.

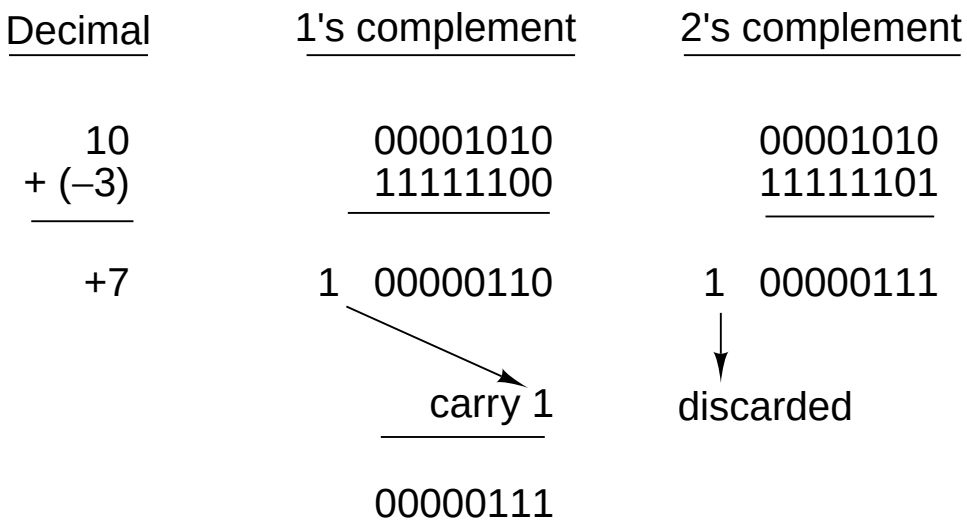


Figure A-9. Addition in one’s complement and two’s complement.

B

FLOATING-POINT NUMBERS

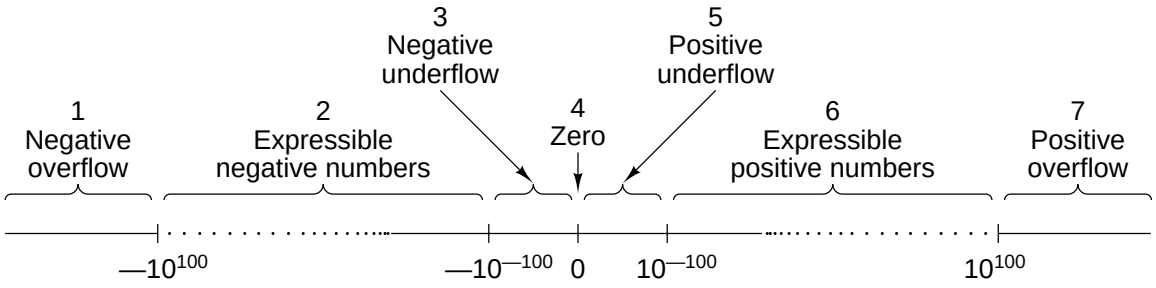


Figure B-1. The real number line can be divided into seven regions.

Digits in fraction	Digits in exponent	Lower bound	Upper bound
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Figure B-2. The approximate lower and upper bounds of expressible (unnormalized) floating-point decimal numbers.

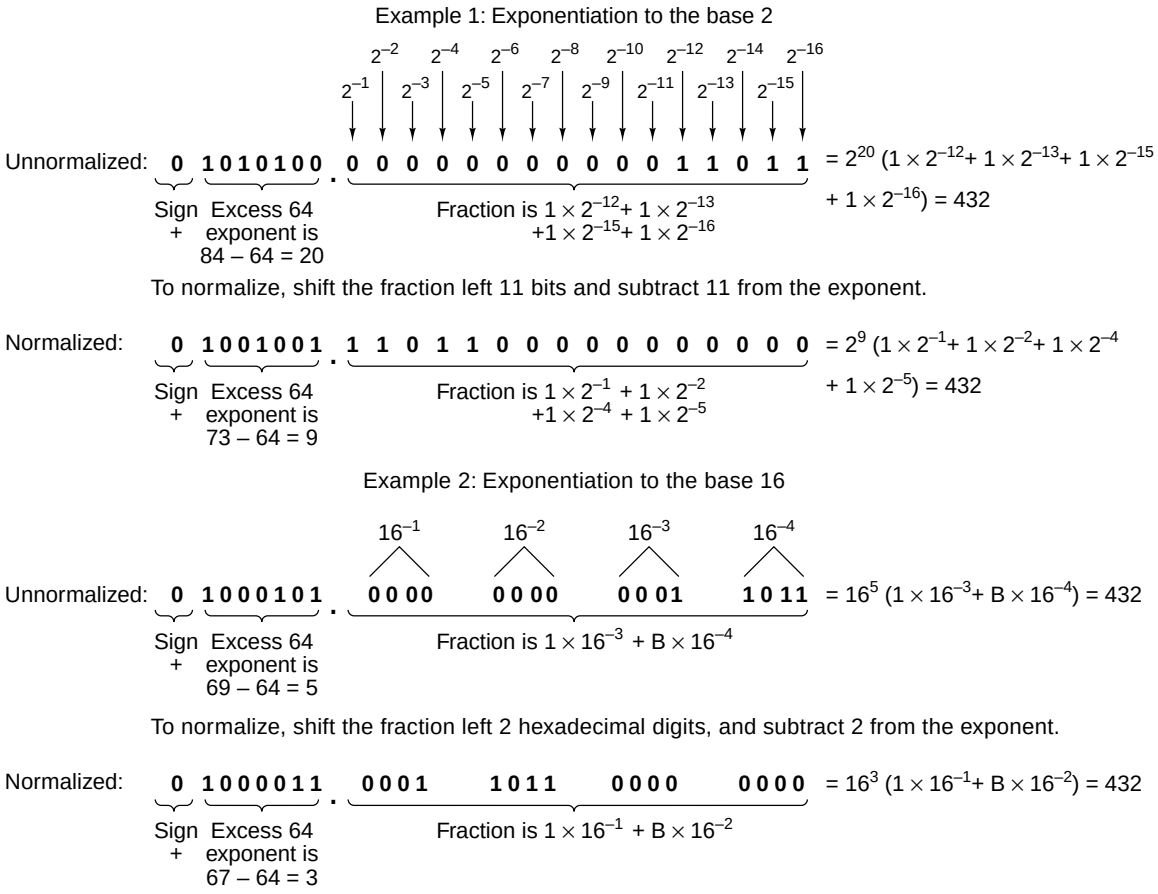
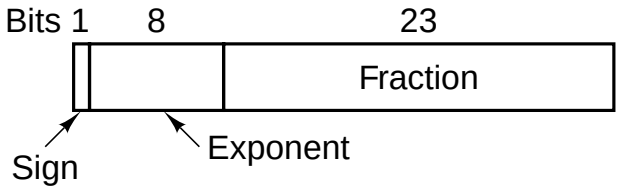
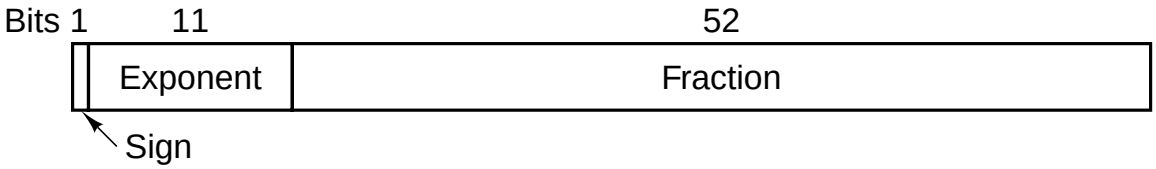

Example 1: Exponentiation to the base 2
Unnormalized: 0 1010100 . 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1
= 2^20 (1 × 2^–12 + 1 × 2^–13 + 1 × 2^–15 + 1 × 2^–16) = 432
Sign + Excess 64 exponent is 84 – 64 = 20
Fraction is 1 × 2^–12 + 1 × 2^–13 + 1 × 2^–15 + 1 × 2^–16
To normalize, shift the fraction left 11 bits and subtract 11 from the exponent.
Normalized: 0 1001001 . 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0
= 2^9 (1 × 2^–1 + 1 × 2^–2 + 1 × 2^–4 + 1 × 2^–5) = 432
Sign + Excess 64 exponent is 73 – 64 = 9
Fraction is 1 × 2^–1 + 1 × 2^–2 + 1 × 2^–4 + 1 × 2^–5
Example 2: Exponentiation to the base 16
Unnormalized: 0 1000101 . 0000 0000 0001 1011
= 16^5 (1 × 16^–3 + B × 16^–4) = 432
Sign + Excess 64 exponent is 69 – 64 = 5
Fraction is 1 × 16^–3 + B × 16^–4
To normalize, shift the fraction left 2 hexadecimal digits, and subtract 2 from the exponent.
Normalized: 0 1000011 . 0001 1011 0000 0000
= 16^3 (1 × 16^–1 + B × 16^–2) = 432
Sign + Excess 64 exponent is 67 – 64 = 3
Fraction is 1 × 16^–1 + B × 16^–2

Figure B-3. Examples of normalized floating-point numbers.



(a)



(b)

Figure B-4. IEEE floating-point formats. (a) Single precision. (b) Double precision.

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	–126 to +127	–1022 to +1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

Figure B-5. Characteristics of IEEE floating-point numbers.

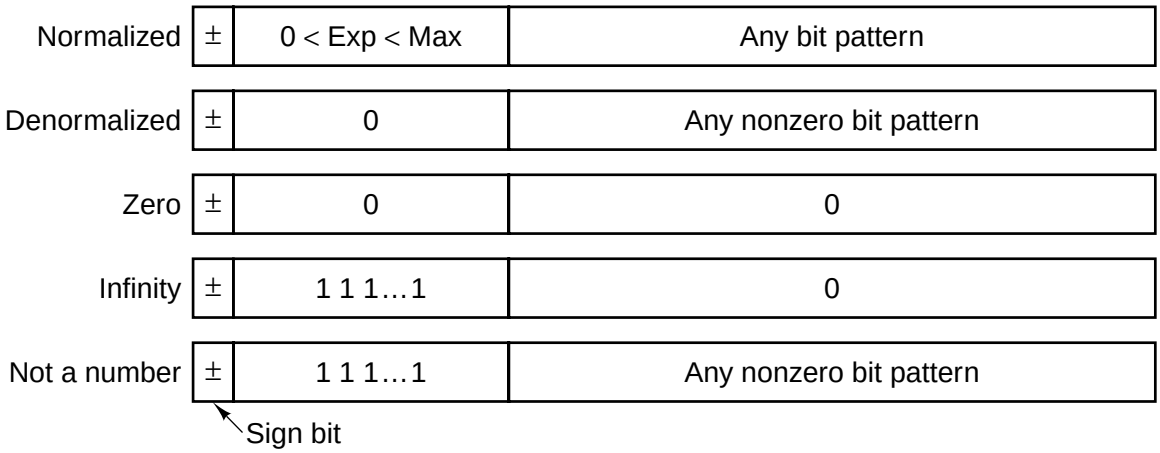


Figure B-6. IEEE numerical types.